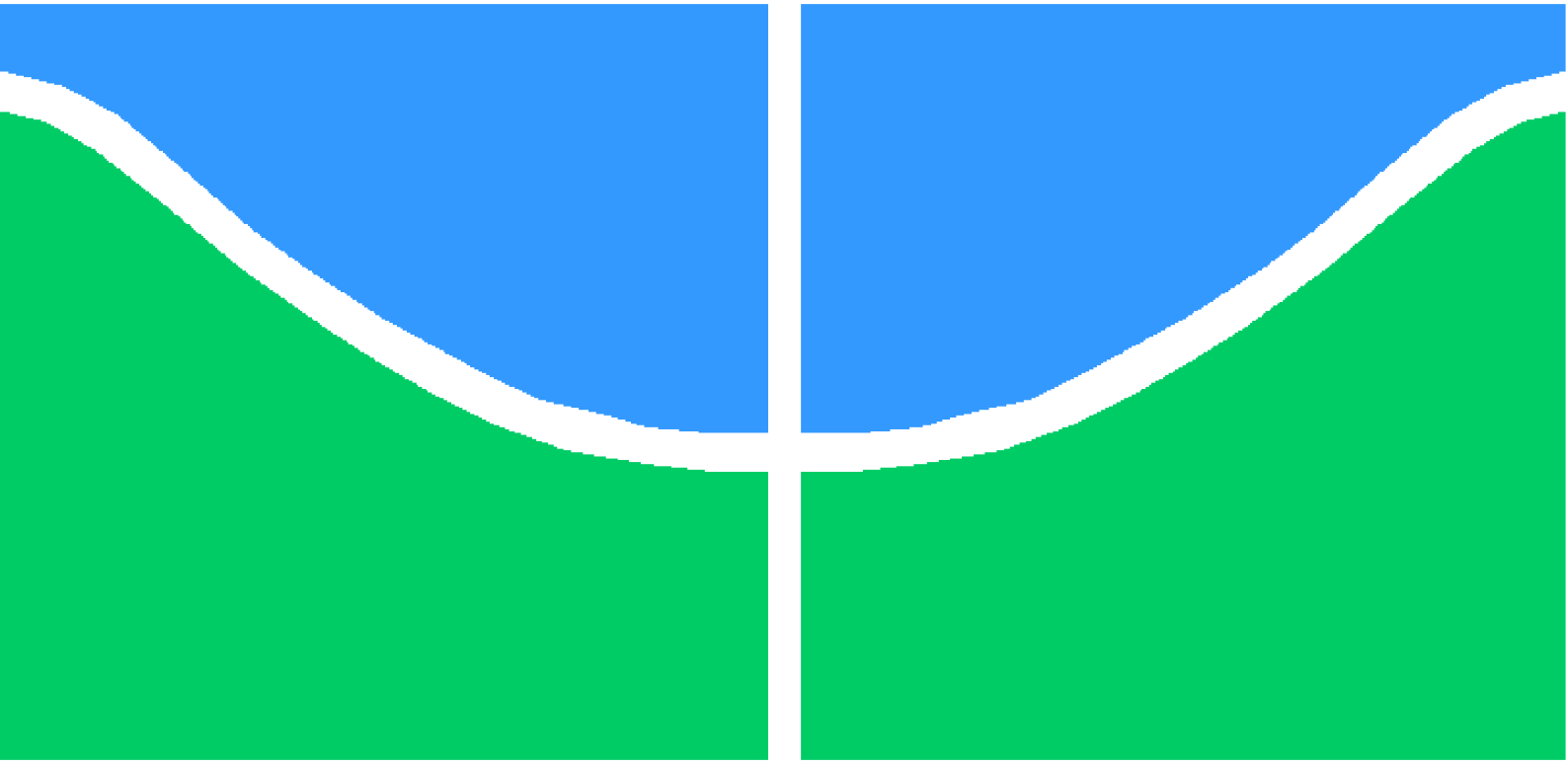


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia Eletrônica

# DESENVOLVIMENTO DE SOFTWARE EMBARCADO AUTOMOTIVO ADERENTE AO PADRÃO AUTOSAR

Autor: Bruno Santiago de Souza da Silva

Orientador: Prof. Dr. Evandro Leonardo Silva Teixeira

Brasília, DF

2014

Bruno Santiago de Souza da Silva

# DESENVOLVIMENTO DE SOFTWARE EMBARCADO AUTOMOTIVO ADERENTE AO PADRÃO AUTOSAR

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Evandro Leonardo Silva Teixeira

Brasília, DF

2014

Bruno Santiago de Souza da Silva
DESENVOLVIMENTO DE SOFTWARE EMBARCADO AUTOMOTIVO ADERENTE AO PADRÃO AUTOSAR/ Bruno Santiago de Souza da Silva. – Brasília, DF, 2014-
84 p. : il.; 30 cm.

Orientador: Prof. Dr. Evandro Leonardo Silva Teixeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. AUTOSAR. 2. MBD. I. Prof. Dr. Evandro Leonardo Silva Teixeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. DESENVOLVIMENTO DE SOFTWARE EMBARCADO AUTOMOTIVO ADERENTE AO PADRÃO AUTOSAR

CDU XX:XXX:XXX.X

Bruno Santiago de Souza da Silva

# DESENVOLVIMENTO DE SOFTWARE EMBARCADO AUTOMOTIVO ADERENTE AO PADRÃO AUTOSAR

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 28 de novembro de 2014:

---

**Prof. Dr. Evandro Leonardo Silva Teixeira**
Orientador

---

**Prof. Dr. Edson Mintsu Hung**
Convidado 1

---

**Prof. Dr. Adson Ferreira da Rocha**
Convidado 2

Brasília, DF
2014

# Agradecimentos

Agradeço em primeiro lugar aos meus pais, Marta Regina e Carlone Batista, por fazer muitos dos meus sonhos se tornarem realidade. Agradeço aos meus irmãos, Carlone Junior e Paulo Santiago, pela grande amizade.

Ao meu Orientador, Evandro Leonardo, pela paciência, dedicação, incentivo e conhecimento que me proporcionou para desenvolver este Trabalho de Conclusão de Curso.

Por fim, a todos os professores, mestres e amigos que me ensinaram, incentivaram e ajudaram, contribuindo assim, para que eu pudesse crescer.

# Resumo

O modelo tradicional de implementação de *software* utilizado na indústria automotiva vem seguindo uma metodologia que desfavorece o rápido desenvolvimento, a redução de custos e a inovação. Pensando nisso, os principais envolvidos nessa área da indústria decidiram realizar um esforço em comum para definir uma arquitetura de *software* padronizada que resolvesse esse problema. Neste sentido o padrão AUTOSAR (*AUtomotive Open System Architecture*) foi proposto com o objetivo de apresentar uma abordagem metodológica para desenvolver módulos de *software* para ECUs (*Electronic Control Unit*). Embora o padrão já esteja definido e especificado, os fabricantes ainda estão receosos em investir neste novo modelo por haver poucas aplicações e estudos de caso. Considerando este cenário o presente trabalho pretende propor uma arquitetura para desenvolvimento de software embarcado automotivo aderente ao padrão AUTOSAR. Para isso foram desenvolvidas as camadas da arquitetura baseadas no padrão e uma aplicação que permite descrever um caso real representando suas ECUs, seus módulos e seu comportamento. A partir desta aplicação é possível gerar todos os códigos necessário para implementar o sistema. Por meio dela foram implementados dois estudos de caso para validar o sistema apresentado. Os resultados obtidos foram utilizados como objeto de analise para avaliar as vantagens e desafios desta metodologia.

**Palavras-chaves**: AUTOSAR, ECU, Unidade de Controle Eletrônica, *Software* automotivo, CAN, J1939.

# Abstract

The traditional model for software implementation used in the automotive industry follow traditional methods that disfavours the fast development, the innovation and the cost cutting. Considering that, the main people involved in this industry area decided to hold a joint effort to define a new architecture of software standardized to solve this problem. In this sense the standard AUTOSAR (Automotive Open System Architecture) has been proposed to present a methodological approach to develop software modules to ECU (Electronic Control Unit). Although the standard has already been defined and specified, the manufacturers are still afraid to invest in this new model due to the few applications for embedded automotive software development that can be attached to the AUTOSAR standard. To solve this many standard, an AUTOSAR based architecture using layers were developed and one application that allows to describe a real case represented by its ECUs, its modules and its behaviour. From this application it is possible to generate all the necessary codes to implement the system. Trough it two case studies were implemented to evaluate the presented system. The results obtained were used as an analysis object to evaluate the advantages and difficulties of this methodology.

**Key-words**: AUTOSAR, ECU, Electronic Control Unit, Automotive Software, CAN, J1939.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| ACK | *ACKnowledgement* |
| ADC | *Analog to Digital Converter* |
| AUTOSAR | *AUtomotive Open System Architecture* |
| CAN | *Controller Area Network* |
| COTS | *Commercial Of-The-Shelf* |
| CRC | *Cyclic Redundancy Check* |
| CSMA/CA | *Carrier Sense Multiple Access / Collision Avoidance* |
| CSMA/CD | *Carrier Sense Multiple Access / Collision Detect* |
| ECU | *Electronic Control Unit* |
| HAL | *Hardware Abstraction Layer* |
| OEM | *Original Equipment Manufacturer* |
| OSEK | *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen*, ou em inglês *Open Systems and their Interfaces for the Electronics in Motor Vehicles* |
| PGN | *Parameter Group Number* |
| RTE | *Runtime Environment* |
| SAE | *Society of Automotive Engineers* |
| VDX | *Vehicle Distributed eXecutive* |
| VFB | *Virtual Functional Bus* |
| XML | *eXtensible Markup Language* |

# Sumário

# 1 Introdução

Segundo Jackman e Sanyanga (2005), o modelo de desenvolvimento de *software* utilizado na indústria automotiva segue uma metodologia que desfavorece a redução da complexidade e a facilidade de integração. No modelo atual, os módulos utilizados nos veículos são desenvolvidos por diferentes fabricantes e existe uma grande dificuldade em fazer com que eles operem facilmente em conjunto, uma vez que são desenvolvidos utilizando diferentes técnicas e padrões. Portanto para conseguir efetivamente integrar os diferentes módulos, as companhias precisam praticamente desenvolver uma tecnologia já existente ao invés de reutilizá-la.

A dificuldade de integrar estes sistemas aumenta ainda mais devido ao crescimento significativo da quantidade de módulos eletrônicos nos veículos, já que segundo Vincentelli e Natale (2007), nos carros mais novos pode-se encontrar até 100 ECUs (*Electronic Control Unit*) interligadas. A necessidade de se adaptar as diferentes técnicas para cada novo modelo de veículo em desenvolvimento levam a um aumento da complexidade, o que por consequência aumenta consideravelmente o tempo de desenvolvimento. Falta portanto, estabelecer um consenso entre os envolvidos para especificar de forma eficiente um conjunto de técnicas e estruturas a serem utilizadas no desenvolvimento dos sistemas eletro-eletrônicos.

Por muito tempo as principais companhias deste setor tentaram construir esta especificação, e a partir deste esforço várias iniciativas surgiram como o OSEK, o OSEK/VDX e o ASAM, que são padronizações abertas para a arquitetura de *software*. Algumas delas apresentaram relativo sucesso, mas nenhuma até hoje foi capaz de unir toda a indústria em torno de uma única metodologia.

A última iniciativa a surgir foi o AUTOSAR. Desenvolvido em conjunto pelos fabricantes de automóveis, fornecedores de componentes automotivos e desenvolvedores de ferramentas, ele procura especificar uma arquitetura de *software* aberta e padronizada para proporcionar uma infraestrutura básica que auxilie no desenvolvimento de *software* embarcado automotivo, além de especificar um modelo de desenvolvimento. Porém, devido à falta de estudos que mostrem a eficiência em adotar esta nova metodologia, este esforço ainda não atingiu os benefícios que eram esperados. Uma vez que, mesmo que o padrão AUTOSAR esteja definido e especificado, os fabricantes ainda estão receosos em investir neste novo padrão.

## 1.1 Objetivo

Diante disso este trabalho tem como principal objetivo propor uma arquitetura para o desenvolvimento de software embarcado automotivo aderente ao padrão AUTOSAR.

## 1.2 Metodologia

Este trabalho adotou uma etapa de pequisa bibliográfica explorando a literatura a cerca de conceitos relacionados a sistemas embarcados automotivos e padrões de desenvolvimento de *software* automotivo. Em uma etapa seguinte foi proposta e implementada uma arquitetura de *software* fundamentada nas camadas do padrão AUTOSAR.

E por fim, foi desenvolvido o software MBDAUTO, que foi construído para realizar a implementação da arquitetura proposta utilizando o desenvolvimento baseado na modelagem das funcionalidades do sistema. Este *software* permite descrever cada um dos módulos de determinado sistema, o comportamento de cada um destes módulos e gerar de forma automática os códigos necessários para implementá-lo.

A partir deste *software* foram construídos dois estudos de caso para realizar a leitura de sensores, a comunicação por meio de um barramento CAN (*Controller Area Network*) e o controle de um painel de instrumentos. O sistema obtido foi objeto de análise para identificar quais os benefícios desta metodologia de desenvolvimento.

## 1.3 Estrutura do trabalho

No Capítulo (2) a revisão bibliográfica é apresentada, ela trata sobre unidades de controle eletrônica (ECU), arquitetura elétrica, protocolos de comunicação e sobre os padrões de desenvolvimento aberto. No Capítulo (3) a metodologia utilizada em cada etapa do trabalho é descrita de maneira geral. Em seguida no Capítulo (4) a arquitetura desenvolvida, a metodologia utilizada em sua construção e os resultados obtidos são detalhados.

No Capítulo (5) é apresentado o *software* MBDAUTO, que foi construído para realizar a implementação da arquitetura proposta e utilizando o desenvolvimento baseado em modelos. No Capítulo (6) dois estudos de caso que foram construídos por meio do software MBDAUTO são apresentados. Por fim, no Capítulo (7) as considerações finais sobre a arquitetura desenvolvida, sua implementação e os resultados obtidos são discutidos.

# 2 Revisão bibliográfica

## 2.1 Eletrônica embarcada

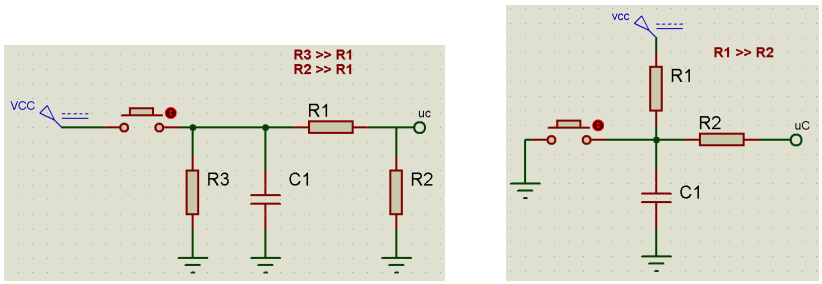
Praticamente todos os carros produzidos atualmente já saem com algum tipo de sistema eletrônico embarcado. Eles são utilizados no controle do sistema de injeção eletrônica, de entretenimento, de aceleração, computador de bordo dentre vários outros.

A importância da eletrônica embarcada nos sistemas veiculares pode ser melhor visualizada a partir do que é apresentado por Cook et al. (2007). Ele afirma que nos carros mais modernos, existem de 20 até 80 diferentes sistemas eletrônicos, sendo estes responsáveis por tarefas que vão desde controles de sistemas com grandes restrições temporais até os sistemas mais simples de um veículo. Cada um destes sistemas é conhecido como módulo eletrônico ou unidade de controle eletrônica.

### 2.1.1 Unidade de controle eletrônica

Segundo Guimarães (2007) a unidade de controle eletrônica (ECU) é responsável pela leitura das entradas, tratamento dos dados lidos, acionamento das saídas e pelo gerenciamento dos protocolos de comunicação que são utilizados no veículo. Cada ECU é composta basicamente de um microcontrolador ou microprocessador que tem seu comportamento descrito por um programa computacional.

As entradas na unidade de controle podem ser digitais ou analógicas. O acionamento das entradas digitais podem ainda ser do tipo *Pull Up*, como é mostrado na Fig. 1a, ou *Pull Down*, como é mostrado na Fig. 1b.

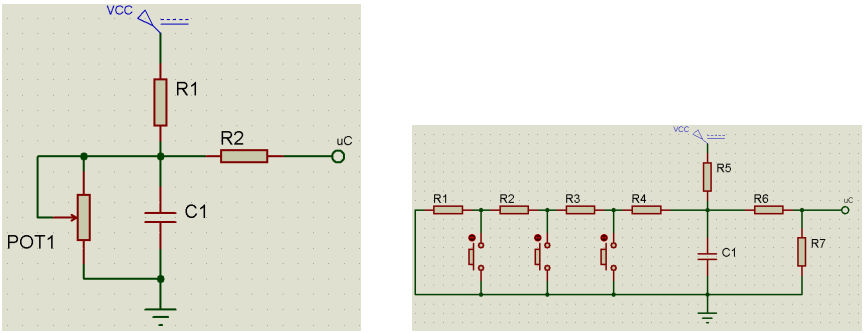



(a) (b)

Figura 1 – a) Acionamento por *Pull Up* ou por sinal positivo. b) Acionamento por *Pull Down* ou por terra.

As entradas analógicas são geralmente utilizadas para a leitura de sensores como é mostrado na Fig. 2a. Também podem ser utilizadas para a leitura de sinais digitais

através de uma rede de resistores, como é mostrado na Fig. 2b.




(a) (b)

Figura 2 – a) Leitura de um sinal analógico. b) Rede de resistores.

As saídas de uma ECU também podem ser do tipo digital ou analógica. Estas saídas podem ser usadas para o controle de dispositivos como LEDs, sirenes, relés, válvulas ou qualquer outro dispositivo que necessite de um acionamento eletrônico. Sendo que a configuração do circuito usado na saída depende do driver interno ao microcontrolador.

O comportamento da ECU é descrito por um *software* que pode ser gravado na memória do microcontrolador ou em uma memória separada. Para as aplicações automotivas ele é divido em três partes. O *firmware* que é um conjunto de instruções programadas diretamente no *hardware* e que descreve o comportamento da ECU, uma calibração básica que contém os valores específicos para cada aplicação e parâmetros programáveis que compreendem bits que podem ser setados ou resetados por dispositivos próprios.

De acordo com Guimarães (2007) existem vários tipo ECUs, diferenciados basicamente pelas funções realizadas e pelas suas características técnicas especialmente em relação ao *hardware*, ele apresenta alguns exemplos, que estão listados a seguir:

- ***Vehicle Control Unit* (VCU):** Unidade de Controle do Veículo.
- ***Multitimer* (MT):** é o módulo mais simples existente, geralmente responsável pela temporização em um veículo.
- ***Body Control Module* (BCM):** Módulo de controle da carroceria. Responsável por todas as funções básicas na carroceria.
- ***Body Electronic Controller* (BEC):** Controlador eletrônico da carroceria. Similar ao BCM.
- ***Front Zone Module* (FZM):** Módulo da área frontal. Geralmente é utilizado quando se decide separar as funções do BCM em vários módulos.
- ***Rear Zone Module* (RZM):** Módulo da área traseira.

- ***Powertrain Control Module* (PCM):** Módulo de controle do motor e transmissão.

Cada uma destas ECUs, principalmente devido a função realizada, costumam ser instaladas em locais específicos do automóvel. Esta separação dos módulos e a necessidade do compartilhamento de informações faz necessário o uso de um protocolo de comunicação específico.

### 2.1.2 Arquitetura elétrica

Segundo Silva, Batista e Varela (2009) a arquitetura elétrica é classificada de acordo com a organização dos módulos interconectados e pela forma de processamento de cada ECU. De modo geral ela é dividida em centralizada ou distribuída.

Na arquitetura centralizada, uma única ECU é responsável por ler todas as entradas, processar as informações e fazer o acionamento das saídas. Não existe a comunicação com outros módulos e nem a necessidade de protocolos de comunicação, pois todo o controle é feito a partir de uma única ECU. A Figura 3 representa o funcionamento desta arquitetura.

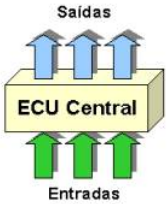



Figura 3 – Arquitetura centralizada.

Fonte: <http://www.pcs.usp.br/~laa/Grupos/EEM/CAN_Bus_Parte_1.html> Acesso em jun. 2014.

Segundo Britto (2008) estes sistemas possuem como vantagem a simplicidade de implementação e manutenção do *hardware*, constituído basicamente pelos sensores, atuadores, ECU, e o cabeamento que os conecta. Como desvantagem, esta configuração exige uma quantidade maior de chicotes elétricos e torna o sistema menos escalável, isto é, dificulta o processo de adicionar novas funcionalidades ao veículo. Além disso, torna mais difícil a reutilização de *hardware* e *software*.

Na arquitetura distribuída se utiliza em um mesmo sistema veicular vários módulos conectados dividindo entre eles as diversas funções existentes no veículo. Neste tipo de configuração é necessário o uso de redes de comunicação entre os diversos módulos presentes. A Figura 4 representa o funcionamento desta arquitetura.

Como apresenta Guimarães (2007), este sistema tem como vantagens a quantidade reduzida de cabeamento, menor tempo de manufatura, maior robustez, maior escalabilidade e maior modularização do projeto. Como desvantagens temos um aumento na complexidade do sistema e consequentemente maior custo de produção.

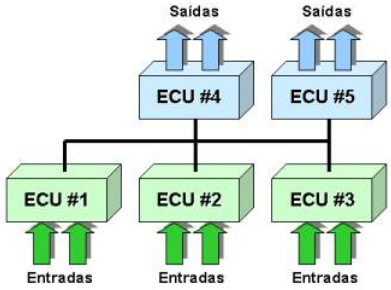
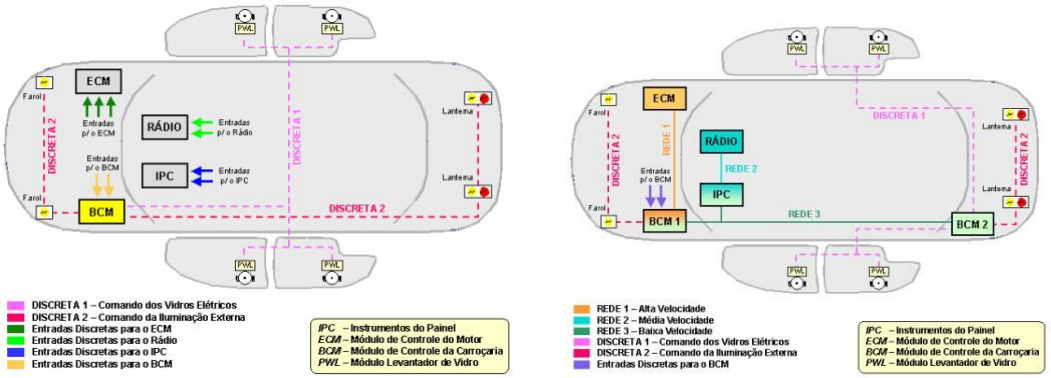



Figura 4 – Arquitetura distribuída.

Fonte: <http://www.pcs.usp.br/~laa/Grupos/EEM/CAN_Bus_Parte_1.html> Acesso em jun. 2014.

Na Figura 5 são apresentados exemplos de utilização dos dois modelos de arquitetura descritos. Na Figura 5a pode-se ver os módulos eletrônicos do ECM (Módulo de Controle do Motor), BCM (Módulo de Controle da Carroceria), IPC (Instrumentos do Painel) e do rádio executando os processos de leitura e acionamento de dispositivos de forma discreta, ou seja, através de uma arquitetura centralizada.

Na Figura 5b é apresentado um exemplo de aplicação utilizando uma arquitetura distribuída onde os mesmos módulos são utilizados interligados por meio de três redes de comunicação e dessa maneira podem compartilhar informações entre si.




(a) (b)

Figura 5 – a) Exemplo de arquitetura centralizada b) Exemplo de arquitetura distribuída.

Fonte: <http://www.pcs.usp.br/~laa/Grupos/EEM/CAN_Bus_Parte_1.html> Acesso em jun. 2014.

Devido as vantagens apresentadas, a arquitetura distribuída é a mais utilizada nos veículos. No entanto esta arquitetura exige que os módulos compartilhem os dados da melhor forma possível e para isso o gerenciamento desta transmissão é realizado por meio dos protocolos de comunicação.

### 2.1.3 Protocolos de comunicação

A quantidade de ECUs (*Electronic Control Unit*) presentes nos veículos faz necessário o uso de mecanismos de gerenciamento que controlem de forma muito eficiente o

compartilhamento de dados no veículo. Para isso existem os protocolos de comunicação.

Segundo Guimarães (2007) os "protocolos de comunicação são meios de transmissão e recepção de dados utilizados para inter-comunicar módulos eletrônicos e/ou sensores e atuadores".

Existem inúmeros protocolos que são utilizados para as mais diferentes aplicações, a SAE (*Society of Automotive Engineers*) apresenta um modo de classifica-los. Tal especificação é determinada em função dos requisitos de comunicação das aplicações. Nesta classificação, as redes se dividem em Classe A, Classe B e Classe C.

De acordo com SANTOS (2010) as redes automotivas de classe A "são redes de comunicação com baixa largura de banda utilizadas em funções de conforto e diagnóstico, como vidro elétrico, retrovisor, controle de bancos, lâmpadas etc". Na Tabela 1 são apresentados exemplos.

Tabela 1 – Exemplos de protocolos de comunicação da classe A

| **Protocolos** | **SINEBUS** | **CCD** | **ACP** | **BEAN** | **LIN** | **SAE J1708** |
|---|---|---|---|---|---|---|
| **Empresa** | DELCO | Chrysler | FORD | TOYOTA | Consórcio | TMC-ATA |
| **Aplicação** | Sistemas de áudio | Controle e diagnostico | Sistemas de áudio | Controle e diagnóstico | Sensore e atuadores inteligentes | Controle e diagnóstico |
| **Meio físico** | Fio único | Fio único | Par trançado | Fio único | Fio único | Par trançado |
| **Codificação dos sinais** | SAM | NRZ | NRZ | NRZ | NRZ | NRZ |
| **Detecção de erros** | Não disponível | 8-bit CS | 8-bit CS | 8-bit CS | 8-bit CS | 8-bit CS |
| **Quantidade de dados** | 10 - 18 bits | 5 bytes | 6 - 12 bytes | 1 - 11 bytes | 8 bytes | — |
| **taxa de transmissão** | 66,6 Kbps - 200 Kbps | 7812,5 bps | 9600 bps | 10 Kbps | 20 Kbps | 9600 bps |
| **Comprimento máximo do barramento** | 10 m | — | 40 m | — | 40 m | 40 m |
| **Quantidade máxima de nós na rede** | — | 10 | 20 | 20 | 16 | 20 |

Fonte: (GUIMARÃES, 2007).

Ainda segundo SANTOS (2010) as redes automotivas de classe B "são redes utilizadas para aplicações importantes para a operação do automóvel e não demandam elevados requisitos de comunicação de dados". Estas redes são aplicadas para conectar ECUs que gerenciam funcionalidades ligadas ao motor, transmissão, embreagem etc. Na Tabela 2 são apresentados exemplos de protocolos pertencentes a esta classe.

Por fim SANTOS (2010) apresenta as redes automotivas de classe C que são "utilizadas em aplicações de segurança crítica com requisitos de tempo real e tolerância a falhas, que estejam diretamente ligadas à dinâmica do automóvel e à segurança ativa". Exemplos deste protocolo são apresentados na Tab. 3.

Tabela 2 – Exemplos de protocolos de comunicação da classe B

| **Protocolos** | **CAN 2.0 ISO-11898 ISO-11519-2** | **CAN 2.0 SAE-J1939** | **J1859 class 2** | **J1850 SCP** | **J1850 PCI** |
|---|---|---|---|---|---|
| **Empresa** | SAE e ISO | SAE | GM | FORD | Chrysler |
| **Aplicação** | Controle e diagnóstico | Controle e diagnóstico | Controle e diagnóstico | Controle e diagnóstico | Controle e diagnóstico |
| **Meio físico** | Par trançado | Par trançado | Fio único | Par trançado | Fio único |
| **Codificação dos sinais** | NRZ | NRZ | VPW | PWM | VPW |
| **Detecção de erros** | CRC | CRC | CRC | CRC | CRC |
| **Quantidade de dados** | 0 - 8 bytes | 8 bytes | 0 - 8 bytes | 0 - 8 bytes | 0 - 10 bytes |
| **taxa de transmissão** | 10 Kbps - 1 Mbps | 250 Kbps | 10,4 Kbps | 41,6 Kbps | 10,4 Kbps |
| **Comprimento máximo do barramento** | 40 m para 1 Mbps | 40 m para 1 Mbps | 35 m | 35 m | 35 m |
| **Quantidade máxima de nós na rede** | 32 | 32 | 32 | 32 | 32 |

Fonte: (GUIMARÃES, 2007).

Tabela 3 – Exemplos de protocolos de comunicação da classe C

| **Protocolos** | **CAN 2.0 ISO-11898 ISO-11519-2** | **CAN 2.0 SAE-J1939** |
|---|---|---|
| **Empresa** | SAE e ISO | SAE |
| **Aplicação** | Controle e diagnostico | Controle e diagnostico |
| **Meio físico** | Par trançado | Par trançado |
| **Codificação dos sinais** | NRZ | NRZ |
| **Detecção de erros** | CRC | CRC |
| **Quantidade de dados** | 0 - 8 bytes | 8 bytes |
| **taxa de transmissão** | 10 Kbps - 1 Mbps | 250 Kbps |
| **Comprimento máximo do barramento** | 40 m para 1 Mbps | 40 m para 1 Mbps |
| **Quantidade máxima de nós na rede** | 32 | 32 |

Fonte: (GUIMARÃES, 2007).

Além destas três existem ainda classes voltadas para funcionalidades específicas como de diagnóstico, *airbag*, direção e freio. Portanto, existem protocolos para atender cada uma das possíveis aplicações dentro de um veículo, dentre eles verifica-se que o que tem a maior variedade de uso é o protocolo CAN.

#### 2.1.3.1 Protocolo de comunicação CAN

Desde os anos 80, muitos sistemas eletrônicos surgiram na indústria automotiva. Segundo Paret (2007) essa evolução se dividiu em três períodos diferentes. No primeiro periodo cada um dos sistemas era independente um do outro, no segundo momento alguns módulos começaram a exigir a criação de um meio de comunicação entre eles e por fim o momento atual, onde os sistemas precisam trocar informação em tempo real.

Essa característica levou a indústria automotiva a desenvolver um protocolo de comunicação para sistemas distribuídos, operando em tempo real e que atendesse a todas os requisitos das companhias. Desse esforço surgiu o protocolo CAN (*Controller Area Network*).

Como é descrito pelo padrão ISO, o CAN é um protocolo de comunicação que suporta de forma eficiente a distribuição de comandos em tempo real com um alto grau de segurança. Seu uso geralmente está relacionado à alta taxa de transferência e aplicações de redes com necessidade de alto grau de confiança na transmissão operando em um cabo multiplexado e de baixo custo.

Segundo Paret (2007) este protocolo tem a finalidade de oferecer uma comunicação serial robusta para conseguir fazer automóveis mais confiáveis, seguros e eficientes e com diagnóstico de falhas.

Uma característica importante da rede CAN segundo Fonseca (2013) "é que ela permite que diversos módulos sejam conectados à linha de comunicação sem que seja necessário um módulo principal para fazer o gerenciamento destas mensagens". A conexão entre estes módulos presentes na rede é feita por meio de um barramento com duas linhas de comunicação como pode ser visto na Fig. 6.

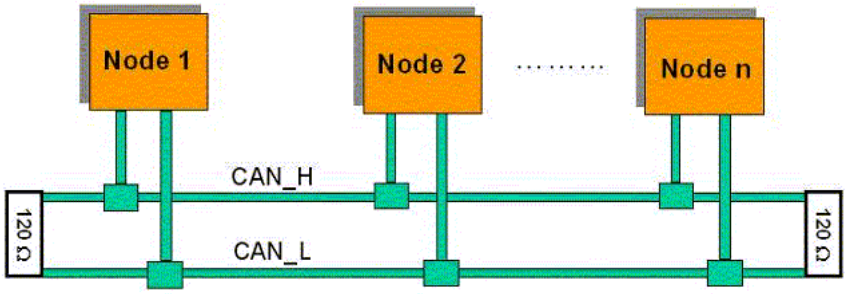



Figura 6 – Linhas de comunicação do protocolo CAN.

Fonte: <http://www.eetimes.com/document.asp?doc_id=1206484> Acesso em jun. 2014.

Os dados enviados através desta rede são interpretados por meio da diferença de potencial entre as linhas de comunicação CAN_H e CAN_L. Essa diferença no barramento indicam se o dado é dominante ou recessivo de acordo com o que é apresentado na Fig. 7.

#### 2.1.3.1.1 Arbitragem

Em sistemas de arquitetura distribuída, é necessário um controle de acesso ao barramento para evitar conflitos quando dois ou mais módulos tentam utilizá-lo. Para que os dados sejam processados em tempo real estes devem ser transferidos rapidamente, e isso exige um meio físico que permita elevada taxa de transmissão e chamadas rápidas à alocação do barramento quando vários módulos tentam transmitir simultaneamente.

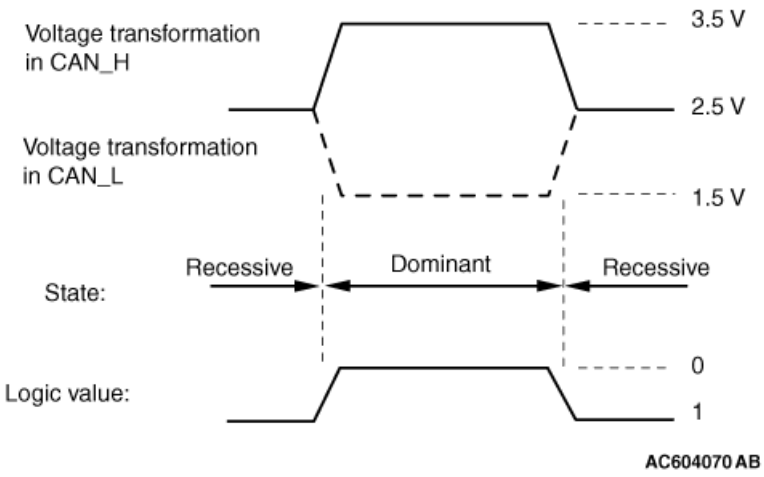



Figura 7 – Bits recessivos e dominantes do protocolo CAN.

Fonte: <http://faq.out-club.ru/download/pajero_iv/maintenance/Service_Manual_2008_2013/2012/54/html/M254200030043200ENG.HTM> Acesso em jun. 2014.

Além disso no processamento em tempo real a urgência da troca de mensagens pela rede pode ser significativamente diferente, uma vez que um dado que varie rapidamente deve ser transmitido com maior frequência e menores atrasos do que outros que variem menos. O protocolo CAN possui um modelo de arbitragem que é capaz de atender a estes requisitos.

Segundo Paret (2007) existem dois princípios diferentes relacionados a arbitragem: CSMA/CD (*Carrier Sense Multiple Access/Collision Detect*) e CSMA/CA (*Carrier Sense Multiple Access/Collision Avoidance*). O procedimento de arbitragem CSMA/CD foi o primeiro a ser desenvolvido para tentar resolver problemas de colisão.

No CSMA/CD quando vários módulos tentam acessar o barramento simultaneamente e ele está ocupado, todas as transmissões são paralisadas e todos os nós saem da rede. Depois de certo período, diferente para cada nó, cada uma deles tenta acessar o barramento novamente. Este modelo de arbitragem não é aconselhável para redes com grandes restrições de tempo e falha como as utilizadas nas aplicações veiculares.

Segundo Guimarães e Saraiva (2002) o protocolo CAN faz uso do protocolo chamado CSMA/CA . Nele um nó que deseja enviar uma mensagem espera até que o barramento esteja ocioso e, em seguida, coloca o primeiro pedaço de quadros identificadores no barramento. Se outros nós tentarem enviar ao mesmo tempo, ocorre um conflito.

Nesse caso, a mensagem que tem a prioridade mais alta ganha o acesso ao barramento. A prioridade da mensagem é definida por um identificador binário (onde o “0” tem maior prioridade). Se houver mais de uma mensagem que tem um “0” na posição do primeiro bit do identificador, o próximo bit é comparado sendo que este procedimento continua por todos os bits do identificador até que a mensagem com maior prioridade

ganhe o acesso ao barramento.

Caso algum nó tenha menor prioridade, este interrompe a transmissão e em determinado momento, tentará enviar a mensagem novamente. Isso é chamado de *run-time scheduling*.

Portanto, a arbitragem do protocolo CAN é não destrutiva, uma vez que a transmissão da mensagem de menor identificador (maior prioridade) não sofre atraso. O acesso ao barramento obedece a prioridade, permitindo que a informação mais urgente seja atendida em primeiro lugar. A retransmissão automática de uma mensagem é feita após a perda no processo de arbitragem.

#### 2.1.3.1.2 Formato da mensagem

Existem dois padrões de mensagens no protocolo CAN, o 2.0A e o 2.0B. Ambos têm sua estrutura dividida em sete campos:

- O início da mensagem;
- O campo de arbitragem;
- O campo de controle;
- O campo de dados;
- A sequência CRC;
- O campo ACK (*acknowledgement*);
- Fim da mensagem.

No padrão 2.0A (mensagem padrão) o identificador da mensagem tem um comprimento de 11 bits (Fig. 8). Para o Padrão 2.0B (mensagem estendida) o identificador da mensagem tem um comprimento de 29 bits (Fig. 9).

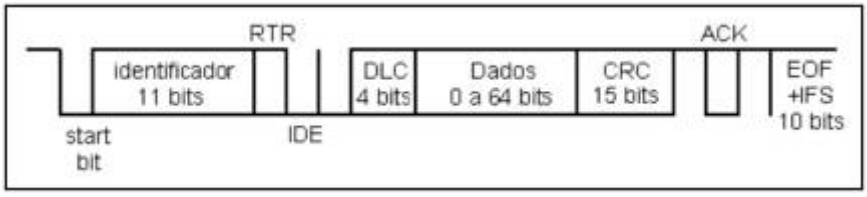



Figura 8 – Formato da mensagem CAN 2.0A.

Fonte: <http://www.pcs.usp.br/~laa/Grupos/EEM/CAN_Bus_Parte_2.html> Acesso em jun. 2014.

Cada uma dos dois padrões citados anteriormente podem transmitir quatro formatos distintos de mensagem. Estes formatos são a mensagem de dados, a remota, a de erro e a de sobrecarga.

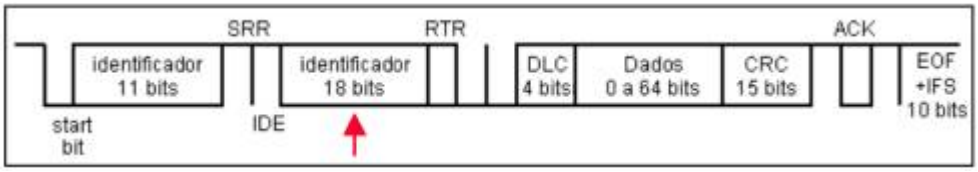



Figura 9 – Formato da mensagem CAN 2.0B.

Fonte: <http://www.pcs.usp.br/~laa/Grupos/EEM/CAN_Bus_Parte_2.html> Acesso em jun. 2014.

Uma mensagem de dados contém os dados do emissor para o receptor. Neste tipo de mensagem estão presentes todos os sete campos que foram apresentados anteriormente, sendo que o bit RTR (*Remote Transmission Request*) deve conter um bit dominante, indicando que o *frame* é de dados.

Um nó que seja receptor de determinado dado, pode iniciar a transmissão do mesmo, através do pedido ao nó de origem, enviando uma mensagem remota. Ou seja, um pedido de dados. O *frame* remoto é constituído por seis campos. Os campos constituintes são idênticos aos de uma mensagem de dados, com a exceção do valor do bit RTR e da inexistência de campo de dados.

A mensagem de erro é transmitida por qualquer nó quando é detectado um erro no barramento e é constituída por dois campos distintos. O primeiro campo é dado pela sobreposição de *flags* de erro provenientes de diferentes estações. O segundo campo é o delimitador de erro.

A mensagem de sobrecarga contém dois campos de bits: *flag* de sobrecarga e delimitador de sobrecarga. A mensagem de sobrecarga é utilizada para provocar um atraso extra entre uma mensagem de dados ou remota e a mensagem posterior.

#### 2.1.3.1.3 Padrões existentes

De acordo com Guimarães (2007) o protocolo CAN é especificado pela ISO 11898 e pela ISO 11519-2. A primeira trata das redes que trabalham com alta velocidade e a segunda das redes que funcionam com baixa velocidade. Estas duas especificações descrevem as camadas físicas e de dados do protocolo.

Guimarães (2007) ainda apresenta os seguintes padrões:

- **NMEA 2000:** Baseado no CAN 2.0B é utilizado em aplicações navais e aéreas;
- **SAE J1939:** Baseado no CAN 2.0B é utilizado em aplicações automotivas;
- **DIN 9684 - LBS:** Baseado no CAN 2.0A é utilizado em aplicações agrícolas;
- **ISO 11783:** Baseado no CAN 2.0A é também utilizado em aplicações agrícolas.

Para este trabalho, o padrão utilizado será o SAE J1939, que é baseado na especificação CAN 2.0B. Seu principal propósito é tornar viável uma interconexão aberta entre os sistemas eletrônicos, proporcionando arquiteturas padronizadas. É usado em veículos comerciais leves, médios, pesados, *on* e *off-road*.

## 2.2 Padronizações abertas

De acordo com Vincentelli e Natale (2007), nos carros mais novos, podem ser encontradas até 100 ECUs conectadas entre si. Este aumento da quantidade de módulos contribui de forma significativa para o crescimento da complexidade, da dificuldade de integração e do tempo de desenvolvimento.

Como Vincentelli e Natale (2007) descreve, os módulos de diferentes fornecedores são construídos a partir de requisitos simples da interface de comunicação e de performance, porém sem levar em consideração uma definição detalhada de temporização, de propriedades de sincronização e dos requisitos dos protocolos de comunicação. Segundo ele a falta de entendimento a respeito de como todos os módulos interagem faz do trabalho de integração uma tarefa desafiadora, sendo que por vezes é necessário praticamente desenvolver uma tecnologia já existente a fim de que todos os sistemas funcionem perfeitamente em conjunto.

Além disso, a complexidade dos sistemas existentes, a necessidade de fazer com que operem perfeitamente em conjunto e a necessidade de reformular detalhes do projeto durante seu desenvolvimento ou até mesmo após sua conclusão refletem significativamente no tempo de desenvolvimento.

Segundo Abowd e Rushton (2002) um veículo demora em média mais de 4 anos para ser desenvolvido, sendo que as definições dos sistemas elétricos, eletrônicos e de *softwares* são feito nas etapas inicias. Essa necessidade de definir as tecnologias e metodologias nas etapas iniciais e de implementá-las durante o processo de desenvolvimento do veículo, faz com que muitas vezes os automóveis sejam lançados já com sistemas obsoletos.

O ideal seria que a integração dos sistemas fosse feita somente nas etapas finais de projeto, mas para isso tanto as ECUs como o *software* necessitam de uma especificação muito bem definida e padronizada. Esta especificação facilitaria o processo de desenvolvimento, uma vez que a independência entre o *software*, o *hardware* e o desenvolvimento do veículo em si possibilitaria a inserção de tecnologias mais novas nas etapas finais dos projetos dos veículos.

A necessidade de se construir sistemas cada vez mais complexos e em um tempo reduzido faz necessário o uso de metodologias que, segundo Vincentelli (2000), descrevam o sistema em um nível de abstração mais alto, que favoreçam a reutilização e que

possibilitem a rápida detecção de erros. Para ele a melhor forma de garantir que o desenvolvimento apresente estas características é definindo uma padronização rigorosa para cada etapa do processo e para a arquitetura utilizada.

No modelo atual de desenvolvimento, para construir várias configurações em um mesmo veículo, elementos de *software* precisam passar por mudanças. Para uma padronização aberta essa variedade de configurações não seria um problema, assumindo que as suas interfaces estariam rigorosamente bem definidas.

Segundo Abowd e Rushton (2002) um padrão aberto consiste de um sistema não proprietário que implemente padronizações abertas suficientes para interfaces, serviços e formatos a fim de permitir que os componentes de engenharia possam ser utilizado para uma grande variedade de meios a partir de uma quantidade mínima de mudanças, permitir que o sistema opere com outros componentes e permitir a interação com o usuário em um estilo que favoreça a portabilidade.

Uma das principais vantagens do uso de uma padronização está na especificação e definição de suas interfaces. Esta característica permite desvincular o desenvolvimento entre o sistema físico, o de *hardware* e o de *software*, facilitando a integração de todos os sistemas. Essa facilidade permite integrar os sistemas nas etapas finais do projeto do veículo, contribuindo para que o mesmo seja lançado com tecnologias mais novas.

Além disso, o uso de um padrão permite que componentes já prontos possam ser reutilizados. Essa possibilidade reduz consideravelmente o tempo de desenvolvimento, e permite que os engenheiros se concentrem mais na tarefa de aprimorar a funcionalidade ao invés de desenvolver todo o sistema novamente.

Porém alguns desafios existem. Para que uma padronização aberta seja bem sucedida e bem aceita é necessário que surja a partir de um esforço em conjunto entre todas as partes interessadas da indústria automotiva.

Outro ponto, envolve a questão de assegurar a competitividade dos fornecedores e OEMs (*Original Equipment Manufacturer*). Atualmente, estes setores garantem sua força no mercado mantendo em segredo seus sistemas e tecnologias. Estabelecer uma nova padronização abrirá espaço para novas empresas, mudando drasticamente a maneira de concorrência entre elas. A idéia é que a competição passe a se estabelecer no desenvolvimento das funcionalidades.

Como Jackman e Sanyanga (2005) discutem, ao longo das últimas décadas várias tentativas de criação de padronizações foram realizadas, como o OSEK (*Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen*, ou em inglês *Open Systems and their Interfaces for the Electronics in Motor Vehicles*), o ASAM (*Association for Standardisation of Automation and Measuring Systems*), a ISO-14230 e o AUTOSAR ( *AUTomotive Open System ARchitecture*), algumas apresentaram relativo sucesso e outra

nem tanto. Nas seções seguintes serão apresentados o padrão OSEK e o AUTOSAR.

### 2.2.1 OSEK/VDX

O padrão OSEK foi fundado em 1993 por um grupo de companhias da indústria automotiva alemã e em 1994 foi unido ao VDX (*Vehicle Distributed eXecutive*). Seu objetivo era de especificar uma arquitetura padronizada e aberta para ECUs em sistemas distribuídos dentro de um veículo (JOHN, 1998).

A padronização que ele apresenta consiste de 4 documentos que definem os requisitos para o sistema operacional OSEK/VDX OS, para o sistema operacional OSEKtime, para os recursos de comunicação e para as estratégias de gerenciamento de rede como a configuração e o monitoramento da rede.

A especificação do sistema operacional OSEK/VDX OS (Fig. 10) fornece uma grande quantidade de serviços e mecanismos de processamento, elas servem como base para o controle de execução em tempo real de aplicações concorrentes. A arquitetura deste sistema faz distinção entre três níveis de processo: interrupção, nível lógico e tarefa.

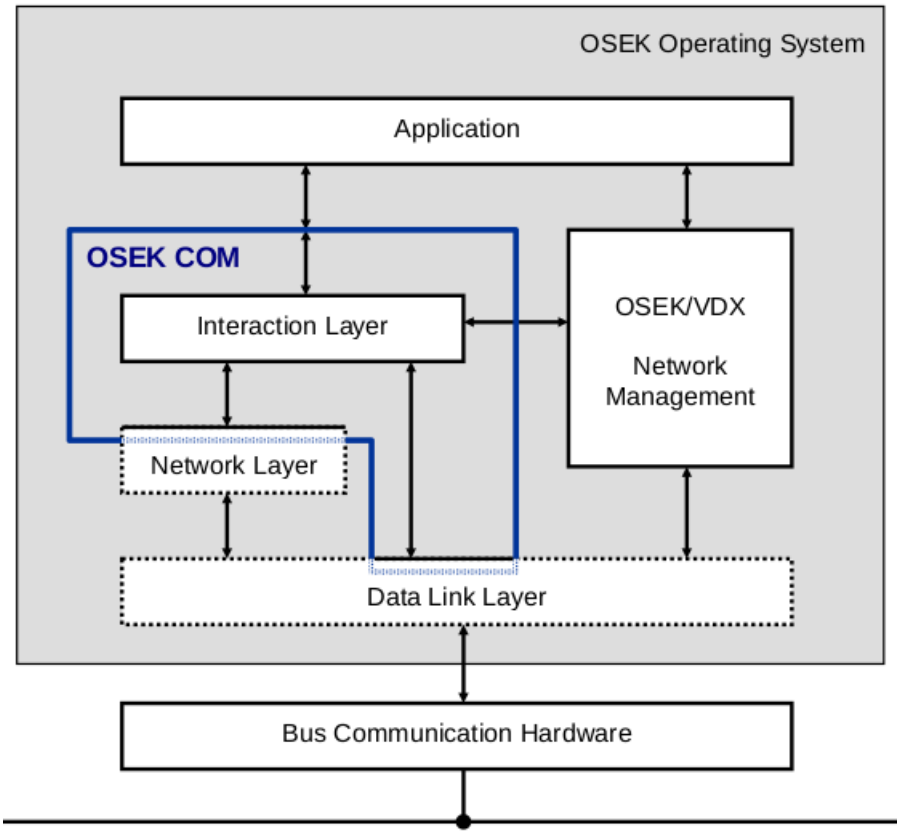



Figura 10 – Arquitetura do sistema operacional OSEK OS.

Fonte: (OSEK-GROUP, 2004).

O sistema operacional OSEKtime (Fig. 11) provê os serviços necessários para aplicações que possuem rigorosas restrições quanto a falhas e funcionamento em tempo real.

Ele é construído de acordo com as configurações especificadas pelo desenvolvedor durante a implementação e não pode ser alterado em tempo de execução. Se a funcionalidade de ambos os sistemas (OSEK/VDX OS e OSEKtime) forem necessárias, é possível executar os dois sistemas paralelamente.

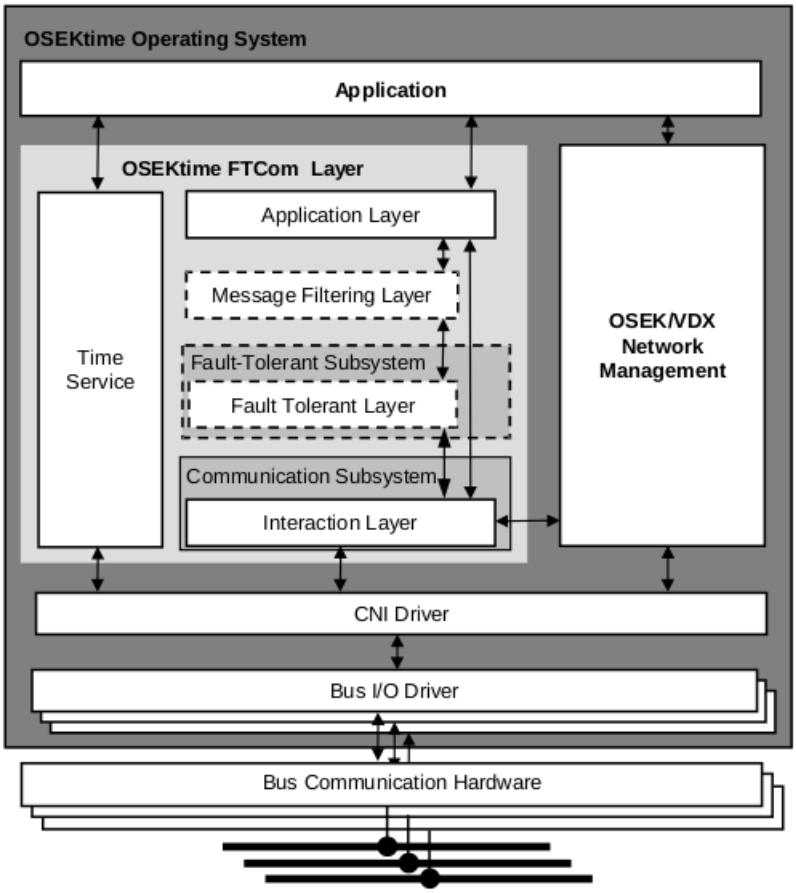



Figura 11 – Arquitetura do sistema operacional OSEKtime.

Fonte: (OSEK-GROUP, 2004).

A especificação dos recursos de comunicação apresenta as interfaces para a transferência de informações na rede do veículo. Essa comunicação pode ser realizada dentro de uma mesma ECU ou entre diferentes ECUs.

O OSEK/VDX permite controlar aplicações distribuídas que são independentes da localização de sua ECU. Como consequência disso o comportamento do sistema local influencia e é influenciado pelo comportamento de todos os sistemas presentes no veículo. Para garantir a confiabilidade e segurança destes sistemas distribuídos, o gerenciamento de rede especificado oferece suporte para vários tipo de tarefas distribuídas.

Além de padronizações quanto a estrutura, também existem definições quanto a maneira de se descrever um sistema OSEK/VDX. Essa padronização ocorre por meio de uma linguagem de implementação própria, chamada de OIL (*OSEK/VDX Implementation Language*)

O uso do OSEK/VDX pode ser bastante vantajoso, uma vez que se implementado corretamente leva à redução do custo, de tempo de desenvolvimento, melhoria na qualidade dos *softwares* produzidos, fornece recursos de interface padronizados para diferentes ECUs e apresenta definições que tornam mais inteligentes o uso dos recursos presentes nas unidades de controle do veículo.

Apesar das vantagens descritas o padrão também apresentou alguns pontos fracos, o maior deles segundo Jackman e Sanyanga (2005) foi a falta de uma padronização para *device drivers*. Essa característica fez com que determinadas aplicações necessitassem de extensos códigos apenas para se comunicar com seus periféricos, sendo que essa é a maior diferença entre ECUs. Dessa maneira, embora o padrão tenha contribuído para aumentar a portabilidade e reusabilidade de *software*, a responsabilidade de desacoplar o *software* do *hardware* utilizado caiu bastante sobre o desenvolvedor.

### 2.2.2 AUTOSAR

A última iniciativa a surgir na indústria foi o AUTOSAR. Ele foi fundado em 2003 com o objetivo de estabelecer uma padronização aberta para os sistemas automotivos e para a metodologia de desenvolvimento. De acordo com Fennel et al. (2006) seu aspecto mais significativo é o de permitir o desenvolvimento de interfaces de *software* padronizadas para qualquer aspecto funcional do veículo. Além disso sua especificação é compatível com outras padronizações como o OSEK e o protocolo CAN.

As discussões iniciais para a criação, definições dos desafios e dos objetivos foram realizadas entre a BMW, Bosch, Continental, DaimlerChrysler e Volkswagen em agosto de 2002. A parceria entre os principais envolvidos foi formalmente assinada em novembro de 2003 e mais tarde outras importantes montadoras entraram para o grupo. Na Figura 12 são apresentados os envolvidos atualmente.

Abaixo são apresentado os objetivos e motivações encontrados no *website* oficial do AUTOSAR (http://www.autosar.org) e que serviram de base para a criação desta padronização.

#### 2.2.2.1 Objetivos

- Comprometimento com os futuros requisitos dos veículos, tais como, disponibilidade e segurança, melhorias, atualizações e manutenção de *software*;
- Aumento da escalabilidade e flexibilidade para integrar e transferir funções;
- Maior uso de componentes COTS (*Commercial of-the-Shelf*) de *softwares* e *hardware* ao longo da linha de produção. “Estes componentes são módulos prontos e fáceis de integrar que são adquiridos de terceiros” (SOUSA; ALENCAR; CASTRO, 1999).

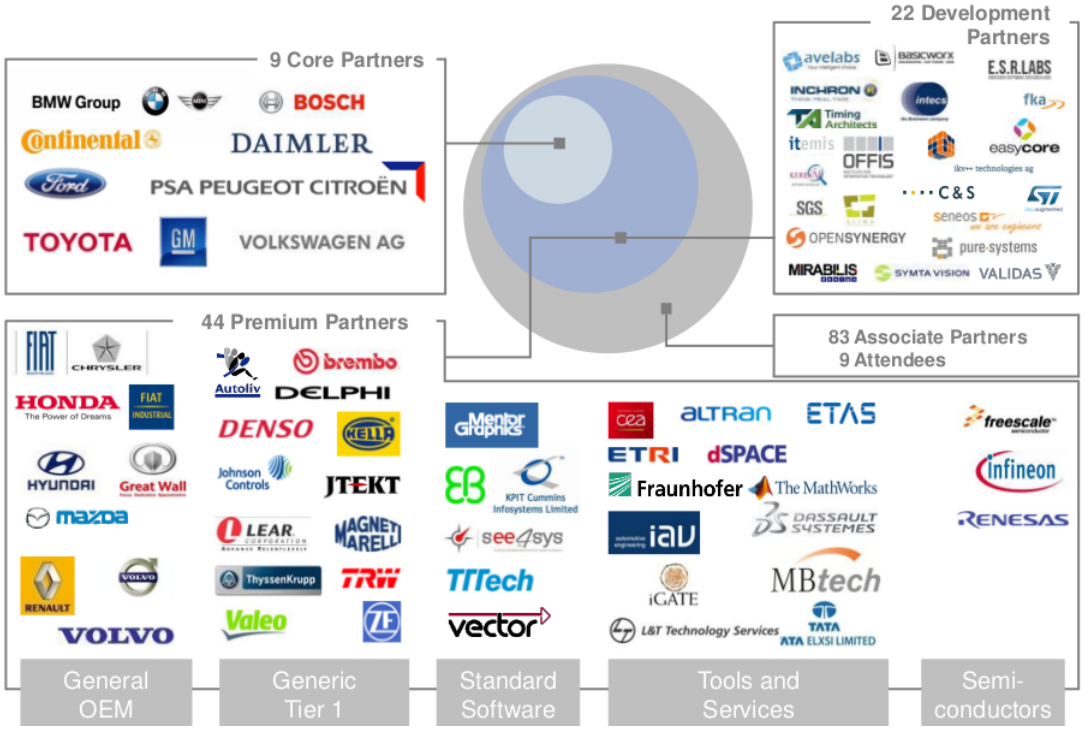



Figura 12 – Membros da iniciativa AUTOSAR.

Fonte: <http://www.autosar.org> disponível em 10 de setembro de 2014.

- Maior controle sobre aspectos como de complexidade e de risco dos produtos e processos;
- Diminuir o custo dos sistemas ao mesmo tempo que os tornem mais escaláveis.

#### 2.2.2.2 Motivações

- Gerenciamento dos sistemas elétricos/eletrônicos quanto ao crescimento da complexidade de suas funcionalidades;
- Flexibilidade para modificações, melhorias e atualizações dos produtos;
- Escalabilidade de soluções;
- Melhorar a qualidade e confiança dos sistemas elétricos/eletrônicos.

#### 2.2.2.3 Visão técnica

O padrão AUTOSAR possibilita o uso de um modelo de desenvolvimento baseado em componentes de *software*. Este modelo é um ramo da engenharia de *software* que enfatiza a separação de módulos funcionais e lógicos com interfaces bem definidas para estabelecer a comunicação entre eles.

Segundo Polberger (2009) uma das principais bases desse modelo é a capacidade de reutilizar um componente já desenvolvido. Dessa maneira, procura-se implementar

componentes independentes e de baixo acoplamento com o sistema. Esta prática tem a capacidade de trazer amplos benefícios a curto e longo prazo para a qualidade do *software* e para os envolvidos no processo de desenvolvimento.

Para ser possível o desenvolvimento baseado em componentes de *softwares*, o AUTOSAR usa uma arquitetura em camadas, como pode ser visto na Fig. 13. Essa característica garante o desacoplamento de funcionalidades do *hardware* e de serviços de *software*.

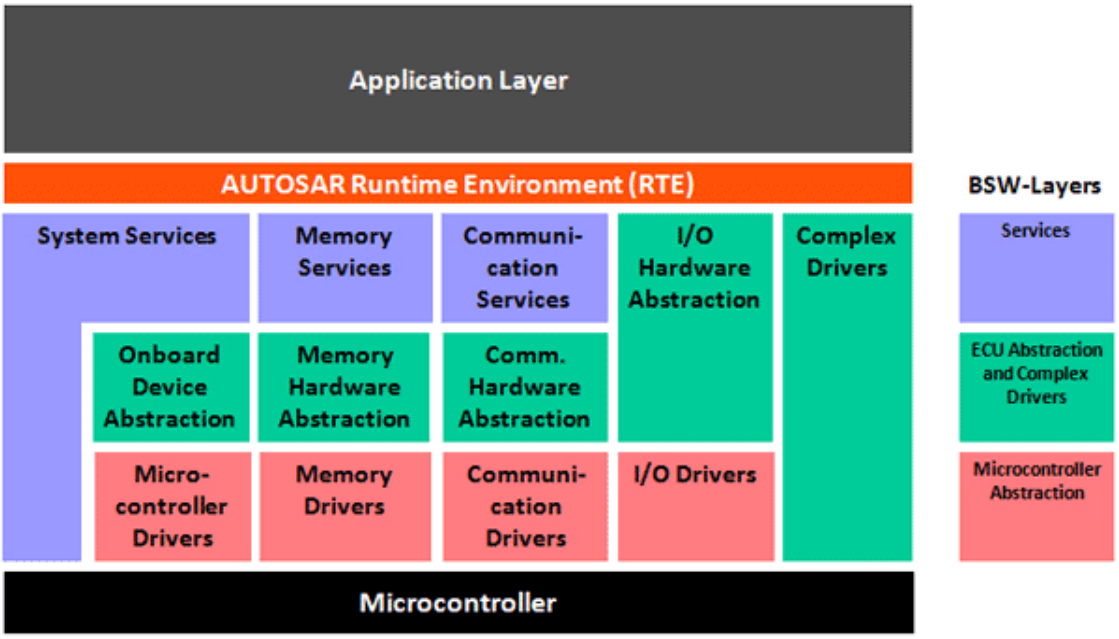



Figura 13 – Camadas da arquitetura AUTOSAR.

Fonte: (FENNEL et al., 2006).

Essa arquitetura é formada por três camadas. A primeira é a camada de aplicação, que é composta pelos componentes de *softwares*. Como Fonseca (2013) descreve, ela é a única que não é composta por um *software* padronizado e é nela em que a funcionalidade está realmente situada.

Cada componente de *software* possui uma funcionalidade que é totalmente independente do *hardware*. Além disso o AUTOSAR não define explicitamente informações a respeito de como cada componente deve ser distribuído entre as ECUs e nem se um componente deve implementar parte de uma funcionalidade ou implementá-la completamente. O que é bem especificado é que pode haver mais de um componente de *software* por ECU e que cada um destes componentes deve estar contido dentro de uma ECU apenas.

Para implementar determinada aplicação, cada componente de *software* precisa conter duas partes: a descrição do componente e a implementação. A descrição do componente de *software*, segundo Fonseca (2013), deve especificar claramente suas interfaces, e também os recursos que serão necessários (como o tempo de processamento da CPU e memória).

Quanto a implementação, esta deve descrever apenas os detalhes relevantes para a funcionalidade e portanto deve ser totalmente independente do *hardware*. Nela é irrelevante se os recursos estão na mesma ECU ou em qualquer outra e se o componente utiliza todos os recursos desta ECU ou se compartilha com outros componentes.

Cada um dos diferentes componentes de *software* são conectados entre si por meio do *Virtual Functional Bus* (VFB). Ele é um componente abstrato que proporciona mecanismos de comunicação e serviços padronizados para estes componentes.

Além de interligar os componentes de *software*, ele é responsável pela troca de informações entre cada módulo. Este meio também permite que todos os serviços de *hardware* e de *software* oferecidos pelo sistema veicular estejam disponíveis, dessa forma os desenvolvedores precisam se concentrar apenas na aplicação, sem aumentar o esforço em codificação e infraestrutura.

Por meio do uso do VFB, a aplicação do *software* não precisa saber com qual outra aplicação irá se comunicar. Basta apenas que o componente disponibilize sua saída para o VFB. Este é responsável por guiar a informação para as portas de entrada do componente de *software* que precisa desta informação.

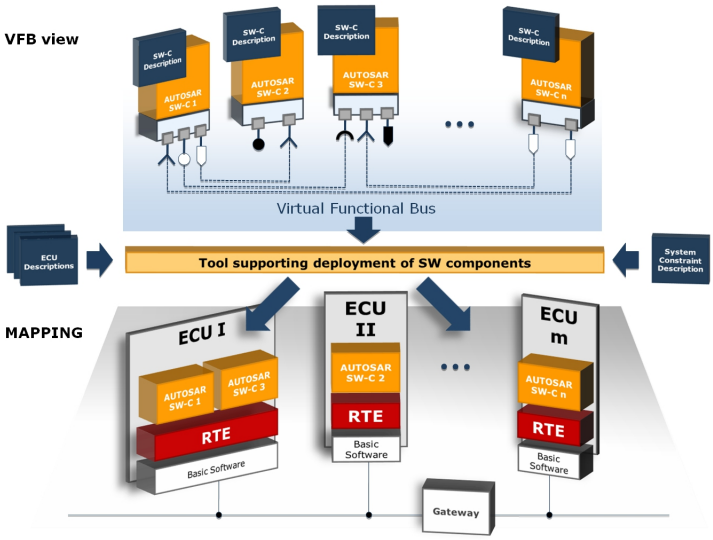



Figura 14 – Relação entre RTE e VFB.

Fonte: (FENNEL et al., 2006).

O VFB abstrai o meio físico de comunicação entre os componentes de *software*, portanto não importa se a comunicação é entre diferentes ECU ou dentro da mesma e nem qual o barramento físico está sendo utilizado, a forma como cada módulo se conecta ao barramento permanecerá inalterada.

Na camada intermediária o padrão especifica o *Runtime Environment* (RTE) e é nesta camada que o VFB é implementado. Segundo Naumann (2009), o RTE consiste de uma representação do conceito do VFB para uma ECU específica. A construção do

RTE ocorre no processo de configuração da ECU que é um passo da metodologia de desenvolvimento do padrão AUTOSAR.

A Figura 14 exemplifica a construção do RTE a partir de um exemplo de aplicação genérica implementado por meio do VFB, onde os componentes da aplicação são distribuídos entre as ECUs existentes e o RTE mapeia a comunicação entre eles.

A última camada é onde está implementado o *basic software* e é nela se encontra o meio que permite abstrair o *hardware*. Como Fennel et al. (2006) apresenta, o *basic software* fornece serviços dependentes e independentes de *hardware* para a camada do RTE e possibilita que ele seja totalmente desacoplado do meio físico.

Uma camada de abstração de *hardware*, segundo Yoo e Jerraya (2003), é todo *software* que é diretamente dependente de um *hardware* específico, mas que cria uma camada padrão para outro *software* que passa a ser totalmente independente deste *hardware*.

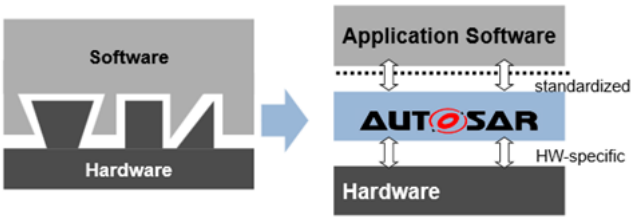



Figura 15 – Representação da camada de abstração de *hardware* do AUTOSAR.

Fonte: (FENNEL et al., 2006).

Para que o *basic software* possa realizar as funções descritas ele é subdivido nas seguintes camadas (Fig. 13):

- Camada de serviços;
- Camada de abstração da ECU e *Drivers* complexos;
- Camada de abstração do microcontrolador.

A camada de serviços provê serviços do sistema, como gerenciamento de memória, protocolos de diagnósticos, gerenciamento de energia e sistema operacional. Excetuando o sistema operacional, os módulos pertencentes à camada de serviço são independentes do *hardware*.

A camada de abstração da ECU é independente do microcontrolador e traduz a ECU para a camada acima. Essa independência é permitida devido à camada de abstração do microcontrolador, a qual contem os drivers específicos do módulo como os de controle das entradas e saídas digitais, conversores A/D e comunicação serial.

Outras partes que lidam com sensores complexos, atuadores com fortes restrições de tempo real e *hardwares* específicos não são padronizados juntamente com o AUTOSAR. Estes módulos em particular estão presentes nos *drivers* complexos da camada de abstração da ECU.

# 3 Metodologia

Este capítulo tem por finalidade apresentar a metodologia adotada ao longo do trabalho. Abordando a realização da pesquisa bibliográfica e as demais etapas de desenvolvimento. Para isso é descrito de forma breve quais os pontos principais de cada etapa e como cada uma foi executada. Em sua estrutura, este trabalho esta divido em 5 etapas conforme é apresentado na Fig. 16, sendo que cada uma está contida em um capítulo diferente.

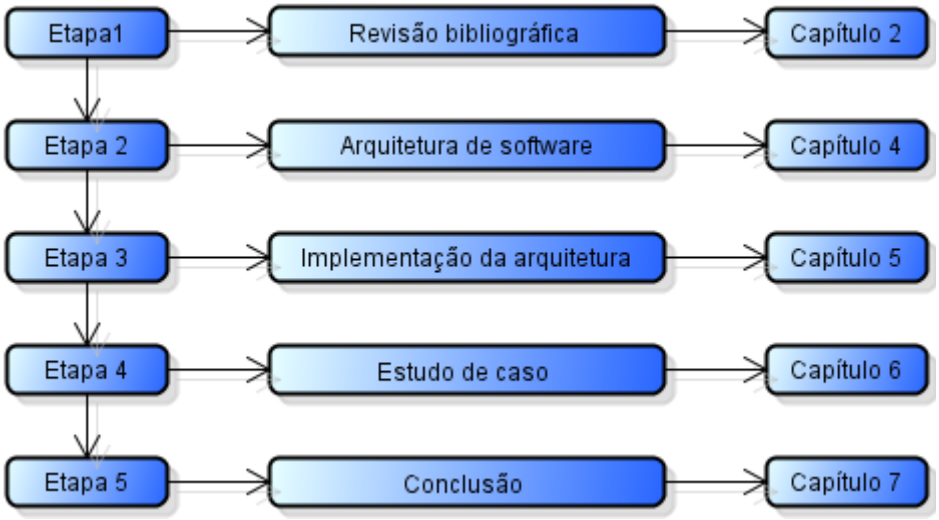



Figura 16 – Etapas realizadas neste trabalho.

## 3.1 Etapa 1

Na primeira etapa foi realizada a pesquisa bibliográfica (Capítulo 2). Esta pesquisa foi executada por meio da exploração da literatura a cerca de conceitos relevantes sobre sistemas embarcados automotivos e sobre padrões de desenvolvimento aberto para *software* automotivo. A partir desta pesquisa foi especificada a arquitetura a ser desenvolvida.

## 3.2 Etapa 2

Nesta etapa a arquitetura proposta foi desenvolvida. Sua construção foi baseada nas camadas do padrão AUTOSAR e a metodologia utilizada consistiu em desenvolver cada uma destas camadas por meio de bibliotecas em C. Cada uma destas biblioteca executa as funcionalidades relativas a respectiva camada. Uma descrição detalhada desta etapa é apresentada no Capítulo 4.

## 3.3 Etapa 3

A terceira etapa deste trabalho teve como objetivo construir o *software* MBDAUTO, que foi feito com o intuito de permitir a implementação da arquitetura desenvolvida por meio de uma modelagem funcional do sistema. Este *software* foi desenvolvido em Java e foi estruturado de modo que pudesse ser totalmente integrado com a arquitetura desenvolvida na etapa anterior. Sua característica mais importante é a de permitir a geração automática de códigos para ECUs (Capítulo 5).

## 3.4 Etapa 4

Para apresentar, testar e validar o sistema proposto foram realizados dois estudos de caso (Capítulo 6). Cada um destes casos foram construídos utilizando o *software* MBDAUTO, que gerou os códigos necessários para a aplicação. O sistema implementado foi utilizado para analisar a eficiência da arquitetura e da metodologia adotada.

## 3.5 Etapa 5

Na etapa final deste trabalho (Capítulo 7) foram utilizados os estudos realizados juntamente com os resultados obtidos na implementação para avaliar o cumprimento dos objetivos, ou seja, propor uma arquitetura para desenvolvimento de software embarcado automotivo aderente ao padrão AUTOSAR, e avaliar as vantagens e os desafios presentes na uso desta arquitetura.

# 4 Arquitetura de *software*

A arquitetura de *software* desenvolvida neste trabalho é baseada nas especificações do padrão AUTOSAR, sendo portanto dividida em camadas de *software*. A Figura 17 ilustra as camadas que foram implementadas. Observa-se que elas foram desenvolvidas por meio de bibliotecas em C que apresentam as funcionalidades de cada nível da arquitetura. Nas seções seguintes cada camada será explicada separadamente abordando seu funcionamento, sua estrutura e a metodologia utilizada para a sua construção.


Figura 17 – Camadas da arquitetura.

## 4.1 *Software* Básico

A camada do *software* básico encapsula a abstração do *hardware*, ou seja, ela é a responsável por traduzir as configurações específicas de cada ECU para a camada do RTE. Seu desenvolvimento é totalmente dependente do *hardware* ao mesmo tempo que dispõe de uma interface bem definida com a camada do RTE. Essa característica permite que as camadas seguintes se tornem independentes do *hardware*.

Para que o *software* básico possa realizar esta funcionalidade é necessário que ele forneça serviços específicos. Nele devem estar contidas as configurações da ECU, serviços de gerenciamento de memória, controle das entradas e saídas digitais, controle dos ADCs (*Analog to Digital Converter*), leitura dos sensores, controlador de comunicação serial e de comunicação CAN. A implementação desta camada da arquitetura foi feita por meio de bibliotecas em C e todos os serviços citados anteriormente foram implementados por meio de funções.

Cada uma destas funções têm seu nome, argumentos e valores de retorno padronizados, de modo que para qualquer ECU todas elas forneçam os mesmo serviços. O que

muda é a implementação de cada serviço que depende da arquitetura e do uso dos recursos do micro-controlador.

Neste trabalho para a construção do *software* básico foram implementadas as funções públicas apresentadas na Tab. 4. Além destas existem ainda funções privadas utilizadas para gerenciar questões específicas da arquitetura e de operação do circuito.

Tabela 4 – Funções presentes na camada do *software* básico.

| Função | Descrição |
|---|---|
| config_ECU()_bsw | Configura o microcontrolador de acordo com suas conexões de *hardware*. |
| **Controlador CAN** | |
| can_reset() | Reseta o controlador CAN. |
| can_set_brp(brp)<br>can_set_prSeg(prSeg)<br>can_set_phaseSeg1(phSeg)<br>can_set_phaseSeg2(phSeg2)<br>can_set_sjw(sjw) | Gerenciam os registradores de *bit timing* do controlador CAN. |
| can_set_mode(mode) | Configura o estado atual do controlador entre os modos: normal, de configuração, *sleep*, *listen* e *loopback*. |
| can_config_reception_objects() | Configura o conjunto de *buffers* que serão utilizados exclusivamente para a recepção das mensagens no barramento CAN. |
| can_set_id(buffer, id, ext) | Configura o registrador do identificador de determinado *buffer*. |
| can_search_for_reception() | Verifica se houve recepção em algum dos *buffers*. |
| can_set_mask(buffer, id, ext)<br>can_filt_and_mask_i(i, id_mask, ext) | Configura os registradores de filtro e máscara de recepção. |
| can_get_id(buffer)<br>can_get_msg(buffer, id, msg, len, stat) | Obtém a mensagem, o identificador e outras informações sobre a mensagem recebida |
| can_find_available_buffer() | Identifica um *buffer* disponível para realizar transmissão |
| can_configure_msg(buffer, len, rtr)<br>can_set_msg(buffer, msg, len) | Configura a mensagem e os registradores para envio de um *frame* CAN |
| can_enable_transmition(buffer) | Habilita a transmissão de uma mensagem. |
| **Comunicação serial** | |
| uart_reset() | Reseta o controlador da comunicação serial. |
| uart_set_baud(baud) | Configuram a velocidade da comunicação serial. |
| uart_send_byte(byte) | Envia um byte pela serial |
| **Leitura dos sensores** | |
| leitura_sensor(id_dado) | Realiza a leitura do sensor especificado |

## 4.2 *Runtime Environment* - RTE

A camada acima do *software* básico implementa as funcionalidades referentes ao *Virtual Functional Bus* e ao *Runtime Environment* do padrão AUTOSAR que fornecem mecanismos de comunicação e serviços padronizados.

Na arquitetura proposta, esta camada é responsável por realizar funções de comunicação com outras ECUs, com sensores, com o painel de instrumentos e serviços ligados a norma J1939.

Cada uma destas funcionalidades são totalmente independentes do *hardware* e são utilizados pela aplicação dentro do componentes de *software*. Os serviços fornecidos por esta camada são construídos de forma independentes devido à padronização da interface com a camada do *software* básico.

A implementação desta camada foi feita por meio de uma biblioteca em C e os serviços citados anteriormente foram implementados em funções separadas dentro desta biblioteca.

Além dos serviços básicos de comunicação e de leitura de sensores, foram implementadas funções que realizam a transmissão no barramento CAN de acordo com a norma J1939. Dessa maneira existem funções para controlar os diferentes aspectos apresentados pela norma: empacotamento, taxa de transmissão e identificação dos pacotes recebidos.

Cada um dos serviços citados anteriormente foram implementados através do conjunto de funções que é apresentadas na Tab. 5.

Tabela 5 – Funções presentes na camada do RTE.

| **Função** | **Descrição** |
|---|---|
| config_ECU() | Executa a configuração da ECU que está implementada no *software* básico. |
| **Protocolo CAN** | |
| can_init() | Reseta o controlador CAN e inicializa todos os *buffers*. |
| can_set_baud(brp, prSeg, phSeg1, phSeg2, sjw) | Configura o a velocidade de comunicação do controlador. |
| can_set_filt_and_mask(id_mask, ext) | Configura os filtros e mascaras dos *buffers* de recepção |
| can_getd(id, msg, len, stat) | Procura por mensagens recebidas nos *buffers* de recepção |
| can_putd(id, msg, len, ext, rtr) | Envia uma mensagem no barramento. |
| can_start_controler() | Inicializa o controlador CAN. |
| **serviços ligados a norma J1939** | |
| can_id(priority, pgn, sourceAdress) | Constrói o identificador da mensagem de acordo com o que é descrito na norma. |
| interrupt_transmission_rate() | Controla a taxa de transmissão de acordo com o *frame* enviado. |
| get_data_J1939(id_frame, id_dado, msg, dado) | Obtém o valor indicado pelo identificador id_dado. |
| update_data( id_dado, dado) | Empacota o dado de acordo com o seu identificador |
| **Comunicação serial** | |
| uart_init(baud) | Inicializa o controlador da comunicação serial e configura sua velocidade de comunicação. |
| **Leitura dos sensores** | |
| sensor(int16 id_dado) | Faz o pedido de leitura do sensor especificado ao *software* básico |
| **Painel de instrumentos** | |
| painel_I(vel, rot, gas, temp)<br>painel_uart_II(valor, id_dado) | Funções que enviam os dados ao painel de instrumentos. |

## 4.3 Componentes de *software*

Os componentes de *software* estão presentes na camada superior da arquitetura. E são através deles que a aplicação é efetivamente implementada. Estes componentes são construído focando totalmente na funcionalidade que se espera realizar, sendo portanto irrelevante informações quanto a configuração do *hardware* ou qual ECU irá recebê-lo.

A aplicação construída nesta camada faz uso das funcionalidades presentes no RTE, favorecendo portanto a reutilização de funções previamente desenvolvidas e tornando o trabalho de desenvolvimento independente de qualquer característica de *hardware*, como está representado na Fig. 18.

Essa característica reduz consideravelmente o tempo de desenvolvimento, e permite

que os engenheiros se concentrem mais na tarefa de aprimorar determinada funcionalidade ao invés de se concentrar em desenvolver código-fonte.

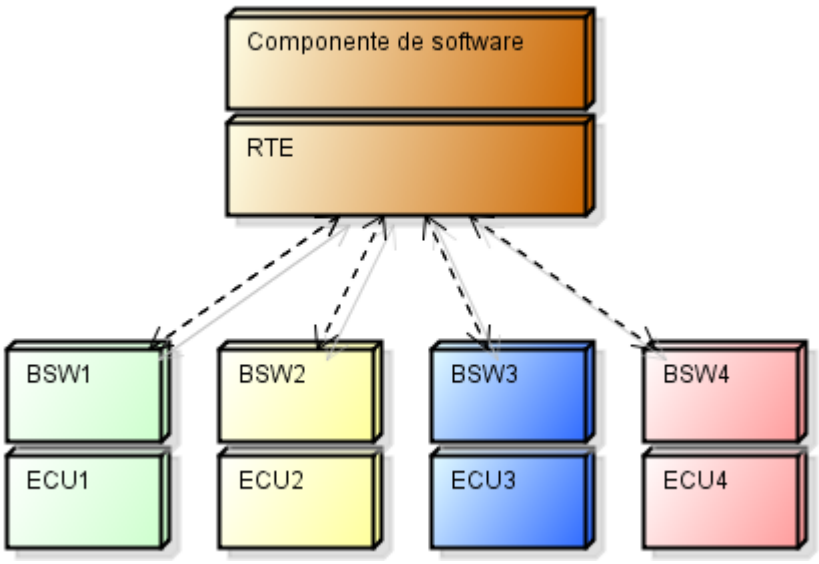



Figura 18 – Independência da arquitetura com relação ao *hardware*.

Para construir a aplicação desta camada foi desenvolvido o *software* MBDAUTO que permite a construção do componente de *software* baseado na modelagem de suas funcionalidades. Nele é possível descrever o sistema que está sendo desenvolvido por meio de diagramas. É através destes diagramas que o código necessário para a implementação do componente de *software* é gerado de forma automática.

# 5 Implementação da arquitetura

Este capítulo tem por objetivo apresentar o *software* MBDAUTO, que foi construído para realizar a implementação da arquitetura proposta utilizando o desenvolvimento baseado em modelos do sistema.

Nas seções seguintes este *software* é apresentado, bem como suas funcionalidades e modo de operação. Em seguida a metodologia utilizada na construção do programa é descrita e o algoritmo que gera o código automaticamente é explicado em detalhes.

## 5.1 *Software* MBDAUTO

Segundo SILVA (2010) o desenvolvimento baseado em modelos "têm como objetivo geral abstrair os detalhes de implementação e de plataforma, focando assim na modelagem das regras do negócio", ou seja, no funcionamento do sistema em si, que deve ser o aspecto mais relevante.

Com o objetivo de permitir o desenvolvimento de componentes de *software* a partir do desenvolvimento baseado em modelos, o *software* MBDAUTO apresenta uma interface onde o usuário implementa, através de diagramas, o funcionamento de determinada aplicação. Sendo que ao final gera todo o código necessário para que cada ECU possa realizar as funções que estão esquematizadas nos diagramas. Para que o usuário possa modelar o sistema, o programa faz uso de um diagrama de ECUs e de dois outros diagramas para cada ECU. O primeiro apresenta o fluxo de dados e o segundo apresenta o diagrama das atividades.

No diagrama de ECUs são representados os módulo presentes no sistema e as conexões dos barramentos de comunicação. Quanto ao diagrama de fluxo de dados, este representa como os sinais transitam dentro de determinada ECU, sendo que ele é constituído de blocos que realizam a recepção, transmissão ou conversão de dados. Um exemplo pode ser visto na Fig. 21, onde os blocos de leitura do sensor fornecem respectivamente os valores da rotação do motor, nível de combustível e velocidade do veículo. Esse valor é então empacotado pelo bloco J1939 que fornece os *frames* para o transmissor CAN.

O diagrama de atividades descreve o comportamento da ECU. Ele é construído utilizando as funcionalidades presentes no RTE e é desenvolvido por meio de um fluxograma. Este fluxograma é utilizado para construir a rotina da função *main* do programa. Para exemplificar observe a Fig. 22, nela estão apresentadas as funções que são utilizadas e qual a sequência lógica de execução destas funções.

A tela inicial do programa é apresentada na Fig. 19. Nesta tela existe uma seção

com um menu onde são mostradas as ECUs que podem ser utilizadas e que são obtidas a partir de um arquivo de configuração. Além deste menu existe outro que lista os diagramas presentes no projeto até o momento.

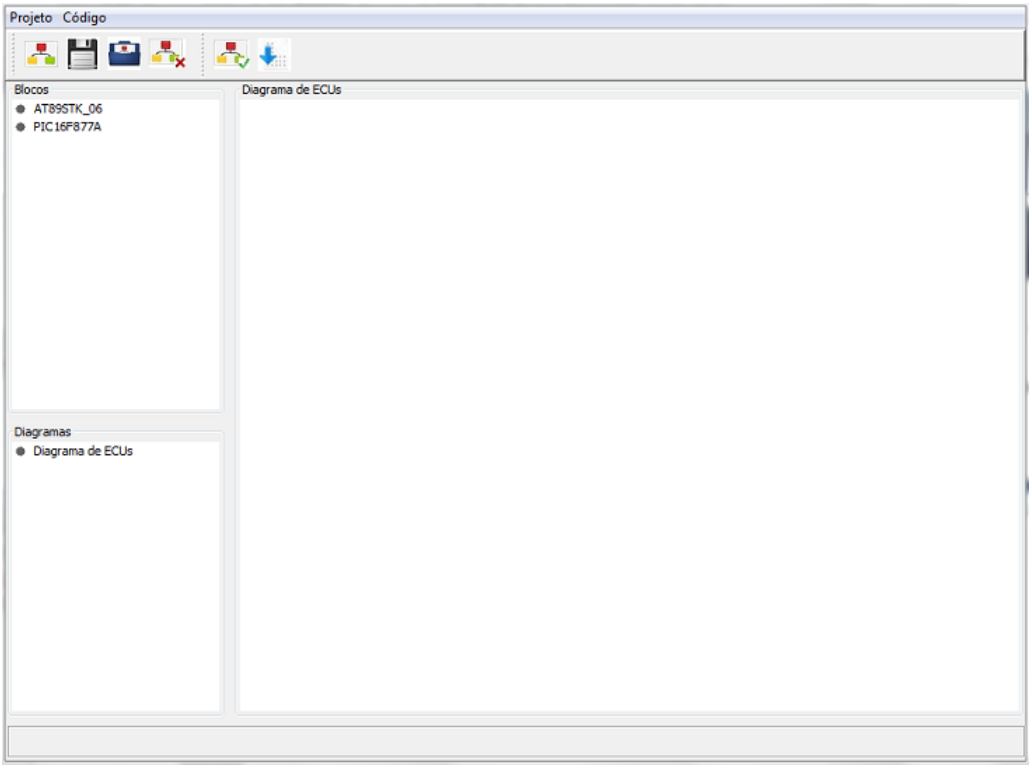



Figura 19 – Tela inicial do *software* MBDAUTO.

Para adicionar novos blocos no diagrama, sejam eles de fluxo de dados, ECUs ou atividades, o usuário deve selecionar o bloco no menu de blocos e em seguida clicar no espaço em branco do diagrama. Para arrastar os blocos deve pressionar o botão direito com o cursor sobre o bloco e arrastar.

Executando um duplo clique sobre uma porta o usuário pode desenhar as conexões entre as portas. Se o duplo clique for em cima de um bloco do diagrama de atividade é aberta uma janela para configurar os argumentos e retorno da função. Clicando com o botão direito em cima do bloco ele é excluído.

No estado inicial do programa o usuário pode construir o sistema de um modo geral por meio do diagrama de ECUs, sendo que para isso deve selecionar o bloco da ECU a ser utilizado, posicionar na tela e fazer as conexões entre os blocos. Conforme são adicionados novos blocos de ECUs, os respectivos diagramas de fluxo de dados e de atividades são listados no menu Diagramas (Fig. 20).

Para navegar entre os diagramas deve-se utilizar o menu que apresenta a lista de diagramas presentes no projeto. Quando um deles é selecionado, o diagrama é aberto e os respectivos blocos são listados no menu Blocos, estes blocos podem ser ECUs, de fluxo de dados ou de atividades dependendo do diagrama selecionado.

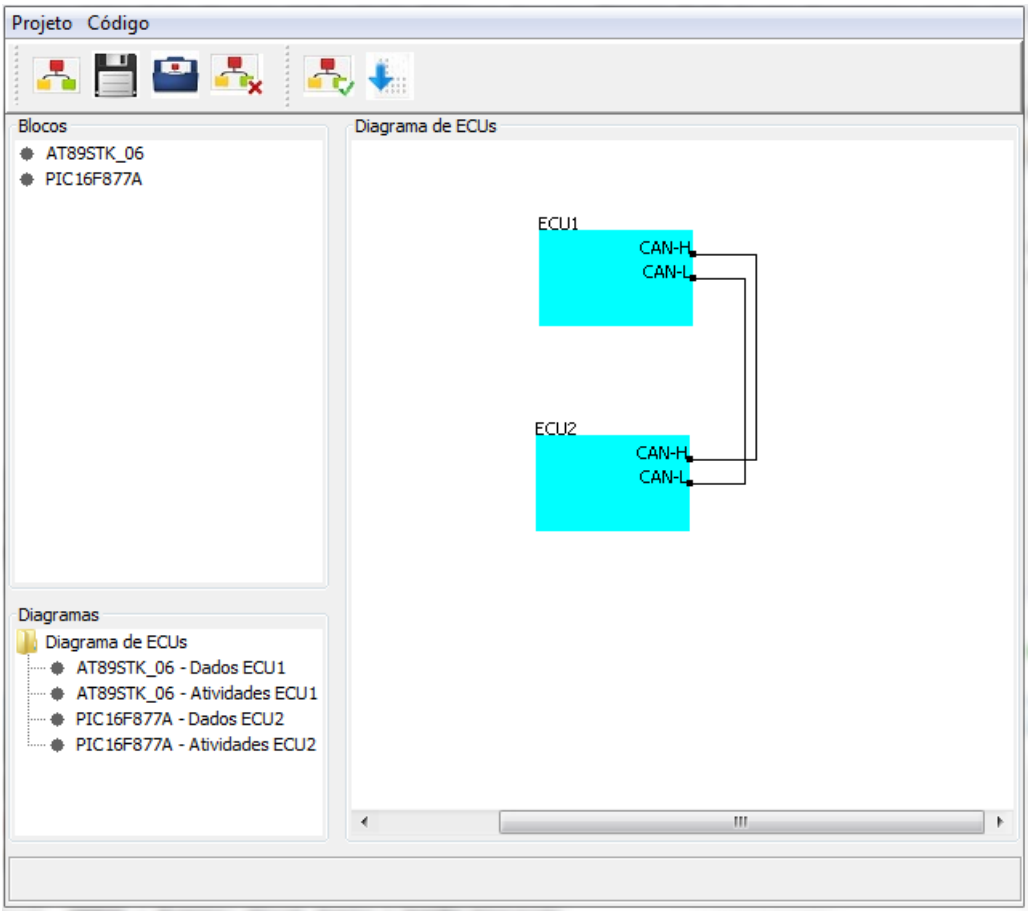



Figura 20 – Construção do diagrama de ECUs.

Cada um destes blocos estão listados dentro de um arquivo de configuração. Desta maneira, para acrescentar novas ECUs, blocos de fluxo de dados, ou funcionalidades presentes no RTE, basta adicionar as informações do bloco no respectivo arquivo de configuração.

Neste arquivo é possível configurar algumas características, como por exemplo quais portas ele deve apresentar no diagrama e, no caso dos blocos do diagrama de atividades, quais são os argumentos e tipos de retorno que determinada função apresenta.

Ao selecionar o diagrama de fluxo de dados de determinada ECU, será aberto o painel para construí-lo, conforme pode ser visto na Fig. 21. Ele deve ser construído de modo que indique caminho percorrido pelos sinais dentro de uma das ECUs do sistema. Em sua construção cada bloco está associado a um conjunto de funções que podem ser utilizadas no diagrama de atividades.

Ao selecionar o diagrama de atividades, será aberto o painel para implementá-lo, conforme pode ser visto na Fig. 22. Ele é construído por meio de um fluxograma que representa as atividades que devem ser executadas dentro de cada ECU, sendo que em sua construção cada atividade corresponde aos serviços presentes no RTE.

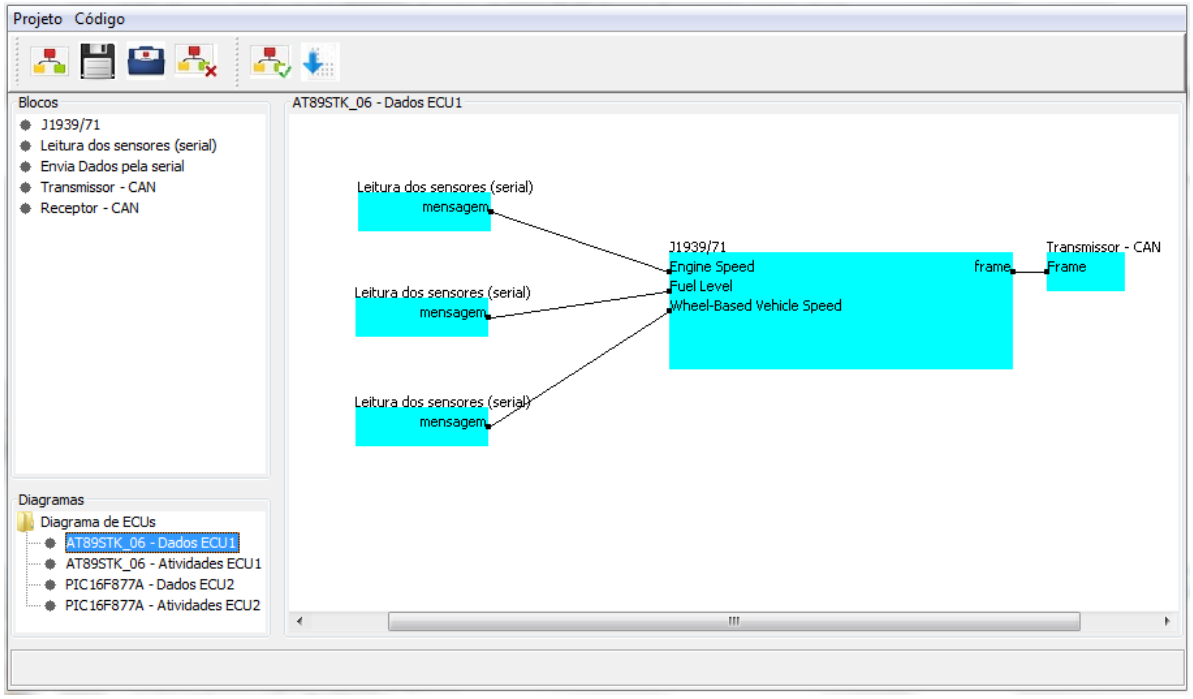



Figura 21 – Exemplo de diagrama de fluxo de dados.

Executando um duplo clique sobre o bloco de atividade é possível visualizar quais são os argumentos que a função possui, qual o seu tipo de retorno e a variável onde o valor de retorno é armazenado. Além disso cada uma destas informações podem ser configuradas conforme a necessidade da aplicação.

Quando ambos os diagramas (diagrama de fluxo de dados e diagrama de atividades) estão devidamente construídos, é possível gerar o código necessário para executar a funcionalidade que está representada nos diagramas.

Este código gerado, tem sua lógica construída a partir do diagrama de atividades, sendo que o algoritmo responsável por construir o código é dividido em duas etapas. Na primeira etapa as conexões entre cada atividade são mapeadas para indicar a ordem e em qual das possíveis ramificações dentro do diagrama esta função está localizada.

Em uma segunda etapa, o algoritmo procura por padrões no mapeamento destas conexões que permitem indicar se as funções ligadas a cada ramo estão dentro da função principal ou dentro de outras estruturas como *while()*, *if()* ou *do-while()*. É durante esta etapa que o código é efetivamente implementado. Um exemplo do código gerado para o diagrama de atividades que é apresentado na Fig. 22 pode ser visto abaixo.

```
#include "config.h"

void main(){
  int16 temp;
  config_ECU();
```

```
  can_init();
  can_set_baud(1,2,7,7,0);
  can_start_controler();
  while(1){
    temp = sensor(65262,175);
    update_ET1(0xFF,0xFF,temp,0xFF,0xFF,0xFF);
  }
}
```

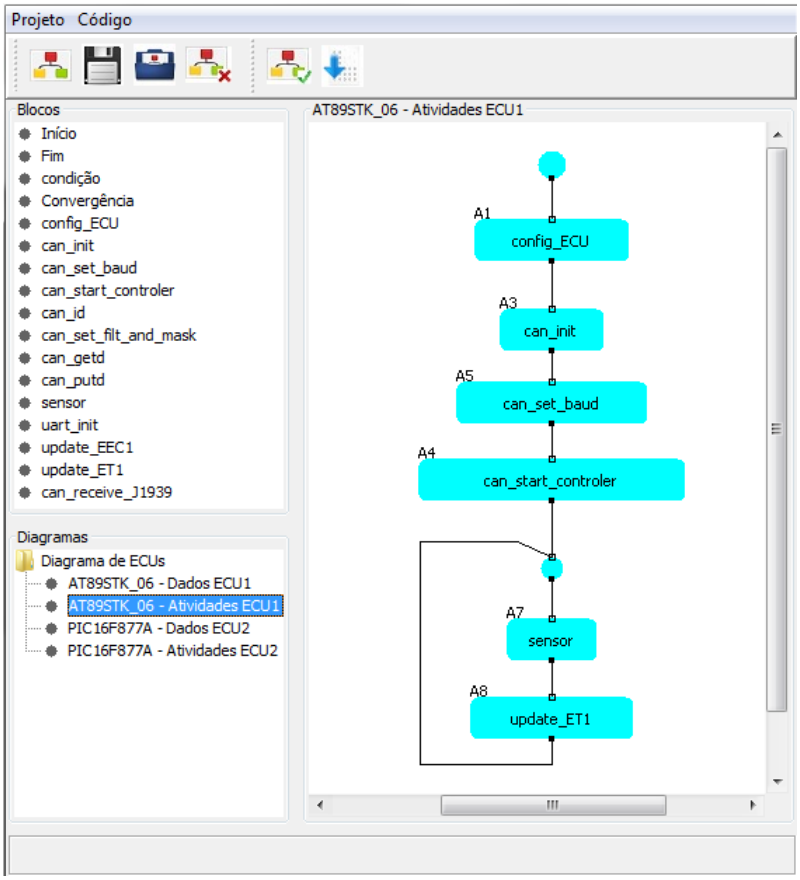



Figura 22 – Exemplo de diagrama de atividades.

Além do código principal também é gerado um outro de configuração que especifica quais serviços do RTE estarão disponíveis para a aplicação. Este arquivo é gerado a partir do diagrama de fluxo de dados, e é construído identificando quais blocos estão presentes neste diagrama e quais as mensagens da norma J1939 foram utilizadas. Um exemplo do código de configuração que foi gerado automaticamente a partir do diagrama da Fig. 21 é apresentado abaixo.

```
#define SENSORES
#define J1939
#define EEC1
```

```
#define DD
#define CCVS
#define CAN
#define CAN_TX
```

## 5.2 Metodologia

O programa MBDAUTO foi desenvolvido por meio da linguagem de programação Java utilizando o ambiente de desenvolvimento NetBeans, que foi escolhido por apresentar recursos que facilitam a criação de interface gráfica e ferramentas de *debug* eficientes.

O código foi estruturado em 3 diferentes pacotes: modelos, telas e controladoras. O pacote de modelos contém o conjunto de classes que abrange todos os objetos necessários na execução do *software*, incluindo blocos, diagramas, conexões, portas e tabelas com configurações.

O pacote de telas é formado por classes que possuem métodos destinados a construir as telas, apresentar suas diferentes configurações, desenhar os diagramas e detectar a ocorrência de eventos.

As controladoras realizam as funções de interação entre as telas, os modelos e os arquivos externos, além de fazer todo o tratamento de eventos. Este pacote é dividido em duas classes, uma destinada a controlar as interações entre os modelos e telas e outra para acessar, editar e criar arquivos externos, portanto é esta classe que faz a escrita do código gerado e a leitura de informações nos arquivos XML, que foram utilizados para armazenar as configurações.

O XML é uma linguagem para a criação de documentos com dados organizados hierarquicamente, tais como textos ou banco de dados. Essa linguagem é classificada como extensível, ou seja, ela pode ser ampliada ou modificada em sua sintaxe e semântica.

Essa é uma característica importante para sua escolha, pois garante que os arquivos possam ser modificados sempre que houver a necessidade de acrescentar novos blocos ou alterar alguma configuração para o uso do programa e para a geração do código.

Por este motivo as informações sobre ECUs, blocos de dados, atividades (funções) presentes no RTE e sobre a norma J1939 foram armazenadas em arquivos XML. Dessa maneira para adicionar novas ECUs ou funcionalidades ao sistema basta acrescentar este novo bloco no respectivo arquivo.

A partir destes arquivos o código busca as informações necessárias para construir o menu de blocos, desenhar os blocos, desenhar suas portas e também as informações

necessárias dos argumentos e retorno das funções para escrever o código. O algoritmo utilizado nesta conversão dos diagramas em código é apresentado na seção seguinte.

### 5.2.1 Processo de geração do código

A utilização do conceito de desenvolvimento baseado em modelos na arquitetura aqui implementada foi possível devido à construção de um algoritmo que realiza a tradução dos diagramas de cada ECU para os respectivos códigos em C. Sendo que cada um dos dois diagramas tem um papel diferente na geração do código.

O diagrama de fluxo de dados é utilizado para gerar um conjunto de *defines* que especificam quais funcionalidades o programa pode usar dentro do RTE. Já o diagrama de atividades representa como o programa deve ser executado e é a partir dele que a lógica presente na aplicação é gerada.

Para que seja possível traduzir este diagrama em um código, é preciso que ele seja construído respeitando algumas regras. De acordo com Gimpel (1980) qualquer algoritmo, por mais complexo que seja, pode ser construído a partir de três estruturas básicas: sequências, decisões e laços (Fig. 23 e 24).

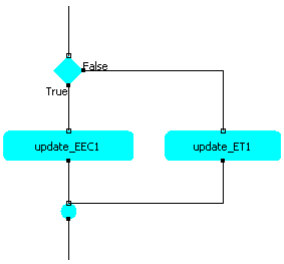
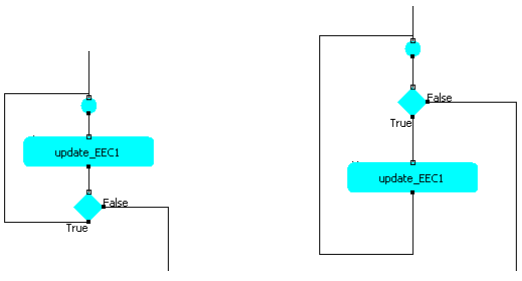



Figura 23 – Exemplo de uma estrutura de seleção.




(a) (b)

Figura 24 – Exemplo de uma estrutura de laço.

Cada uma destas estruturas podem ser combinadas e estar contidas uma dentro da outra indefinidamente, porém elas nunca podem saltar diretamente de dentro de uma estrutura para a outra.

No diagrama aqui implementado a construção de cada estrutura é feita a partir de cinco blocos básicos: início, fim, processo, condição e convergência (Fig. 25). Cada um desses blocos podem ter somente uma conexão por entrada e por saída, com exceção do bloco de convergência que pode ter duas conexões em sua porta de entrada.

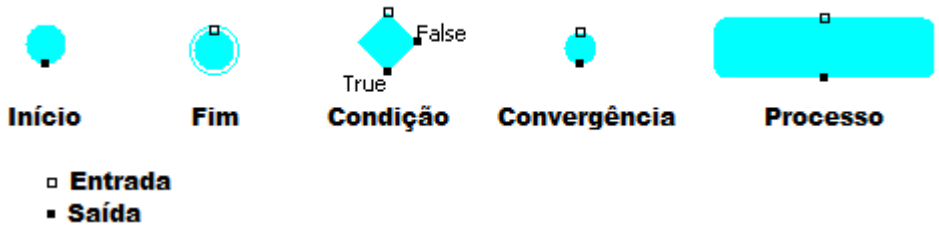



Figura 25 – Blocos básicos do diagrama de atividades.

A literatura aborda a geração automática de códigos a partir de fluxogramas estruturados por meio de diferentes técnicas. Porém dentre estas fontes, várias apresentam restrições significativas em seu uso ou os algoritmos utilizados não são públicos.

Dakhore e Mahajan (2010) por exemplo, aborda uma estratégia de conversão baseada no uso de arquivos XML, porém é necessário que o usuário especifique o fim das estruturas de decisões e de laços, o que prejudica a eficiência do algoritmo.

Já Carlisle et al. (2004) apresenta uma técnica diferente que busca encontrar as estruturas, mas que possui um problema semelhante onde o usuário no momento de construir o diagrama deve usar blocos que indicam onde estão as decisões e onde estão os laços. Além disso o autor não apresenta detalhes do algoritmo.

Neste trabalho foi utilizado um algoritmo semelhante ao apresentado por Wu et al. (2011) que é construído baseado em buscas iterativa que procuram por estruturas básicas (sequências, laços e decisões), como elas estão relacionadas e qual a profundidade de cada uma. No entanto neste algoritmo também é necessário indicar o fim destes blocos.

Para eliminar a necessidade de se indicar as estruturas, foi construído um algoritmo para mapear o diagrama. Através deste diagrama é possível avaliar como ele está estruturado, em seguida este mapeamento é utilizado para escrever o código.

Na etapa de mapeamento as conexões do diagrama são nomeadas de acordo com as ramificações a qual pertencem de modo que é possível identificar o fim das estruturas de laços e decisões. Essa nomeação é constituída de um vetor que possui números indicando quanta estruturas existiram anteriormente e qual foi caminho percorrido através dos blocos de decisão para chegar àquela ramificação.

A Figura 26 exemplifica como funciona este mapeamento. As primeiras conexões estão nomeadas com o número $[1]$ somente, indicando que até o momento só existem processos. Quando existe um bloco de decisão é acrescentado um identificador da ramificação

a qual a conexão pertence, seguido de uma numeração que indica por quantas estruturas o diagrama já passou.

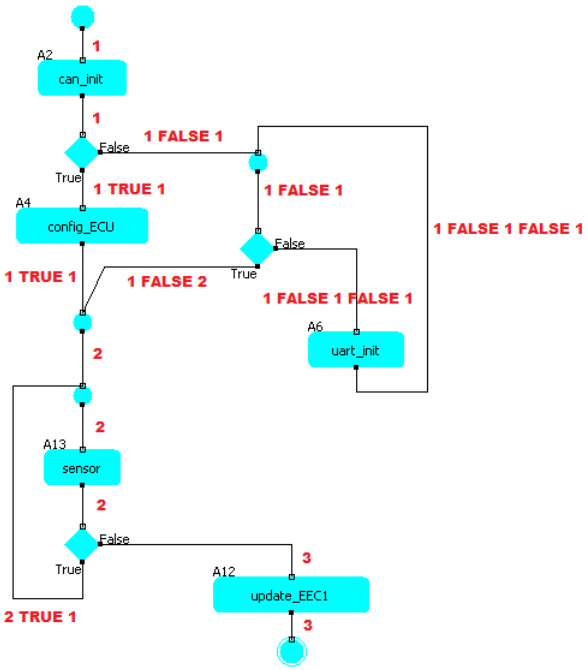



Figura 26 – Exemplo de mapeamento realizado pelo algoritmo.

Como pode ser visto na Fig. 26, a ramificação falsa da condição associada ao bloco A6 possui o mapeamento [*1 FALSE 1 FALSE 1*], essa ramificação volta para um bloco de convergência que está associado a conexões que possuem um mapeamento anterior ao bloco de decisão e isto indica um laço. Já a conexão que está localizada depois do bloco de decisão do laço possui o mapeamento [*1 FALSE 2*], indicando que ela pertence ao ramo falso da primeira condição e que saiu de uma estrutura dentro dessa ramificação.

Isso também ocorre na conexão com mapeamento [*2*] depois da convergência, indicando que essa conexão não pertence a nenhum ramo de decisão e que para chegar a esta conexão é necessário passar por pelo menos uma estrutura.

Utilizando este mapeamento também é possível identificar quando existem estruturas aninhadas, como pode ser visto na Fig. 27, onde existem duas estruturas de decisão uma dentro da outra.

Dessa forma, o algoritmo de mapeamento é construído para executar uma sequência de iterações que buscam passar por todas as ramificações e combinações de caminhos possíveis do diagrama.

Primeiramente o algoritmo passa pelas ramificações verdadeiras colocando todos os blocos de condição em uma pilha até que chegue ao bloco fim ou até encontrar uma conexão já mapeada, o que só acontece quando atravessa um bloco de convergência.

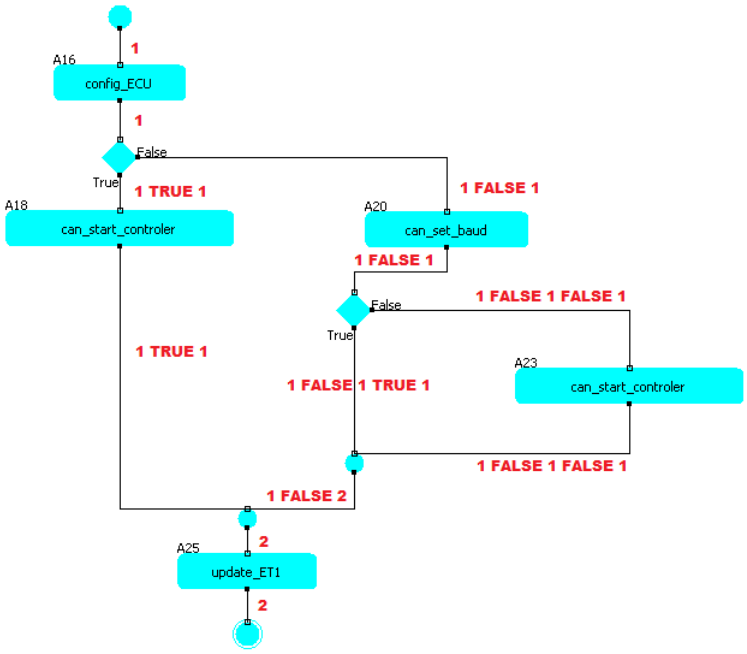



Figura 27 – Mapeamento de estruturas aninhadas.

Se chegar ao fim, o bloco de condição é retirado do topo da pilha e colocado em uma pilha de condições secundária e o algoritmo é executado em sua ramificação falsa. Se chegar a uma conexão já mapeada, a condição também é retirada da pilha mas não é colocada na pilha secundária e o algoritmo continua a partir da ramificação que sai da estrutura.

Quando o algoritmo encontra uma conexão já mapeada novamente ele verifica se a convergência é o final da estrutura de decisão ou se ela é parte de um laço, isso é feito comparando o mapeamento da conexão anterior com o da atual. Se for o final de uma decisão, a condição no topo da pilha secundária é retirada. Baseado nestes resultados, a conexão seguinte à estrutura é identificada e mapeada.

Esse procedimento continua por todas as conexões até que todas as ramificações das condições presentes nas pilhas sejam mapeadas corretamente. Desta maneira o algoritmo percorre todos os caminhos possíveis entre o início e o fim do diagrama atribuindo um conjunto de identificadores que permite reconhecer qual a posição de determinada conexão dentro das possíveis estruturas. Uma visão mais detalhada do algoritmo é apresentada na Fig. 28.

Com o mapeamento pronto, o algoritmo de geração de código percorre o diagrama uma única vez, identificando cada uma das estruturas e escrevendo o código. Para isso ele observa a sequência em que os blocos estão posicionados e o mapeamento das conexões.

A medida que vai avançando ao longo do diagrama as funções especificadas em cada processo são escritas. Quando o algoritmo encontra um bloco de condição, procura

então se existe no topo da pilha de instruções algum *do-while* e se ele está relacionado com este bloco.

Senão este bloco é tratado como um *if*, sendo que a instrução é colocada na pilha de instruções, em seguida a condição configurada no bloco é escrita no código e o algoritmo começa a percorrer o ramo que possui a indicação TRUE no mapa de suas conexões.

O programa continua procurando por padrões e escrevendo as funções presentes nos processos. Se houver qualquer outra estrutura dentro desta ramificação ela será tratada normalmente até que se encontre o bloco de convergência que indica o fim deste *if*, o que é verificado comparando o mapeamento das duas conexões que chegam no bloco.

Após encontrar o fim do *if* ele é retirado da pilha de instruções, então a instrução *else* é escrita e adicionada à pilha. Neste momento o algoritmo passa a percorrer o ramo que possui a indicação FALSE. O programa então continua procurando por padrões e escrevendo as funções presentes nos processos até encontrar a convergência que indica o fim do *else*.

Quando o algoritmo encontra um bloco de convergência ele verifica inicialmente se este é o fim de um *if* ou *else*. Se não for, verifica se é um laço, para isso avalia se o mapeamento da segunda conexão que chega na convergência vem de algum ponto futuro do diagrama.

Caso seja detectado um laço, o algoritmo procura por padrões no arranjo dos blocos conectados à convergência e no mapeamento das conexões que chegam e saem deste bloco e dessa maneira consegue identificar se a próxima instrução é um *do-while*, *while*, ou *while(TRUE)*.

Quando uma dessas estruturas é identificada, a instrução é escrita e adicionada à pilha de instruções juntamente com uma informação sobre qual o bloco final dela (esta informação só existe nos laços). No momento em que o algoritmo alcança um bloco final de determinada instrução ele escreve o fim desta estrutura no código.

Dessa maneira o algoritmo continua percorrendo o diagrama procurando estes padrões e escrevendo o código até encontrar o bloco final do fluxograma ou o fim de um *while(TRUE)*, o que indica o fim da geração do código. Uma visão mais detalhada do algoritmo é apresentada na Fig. 29.

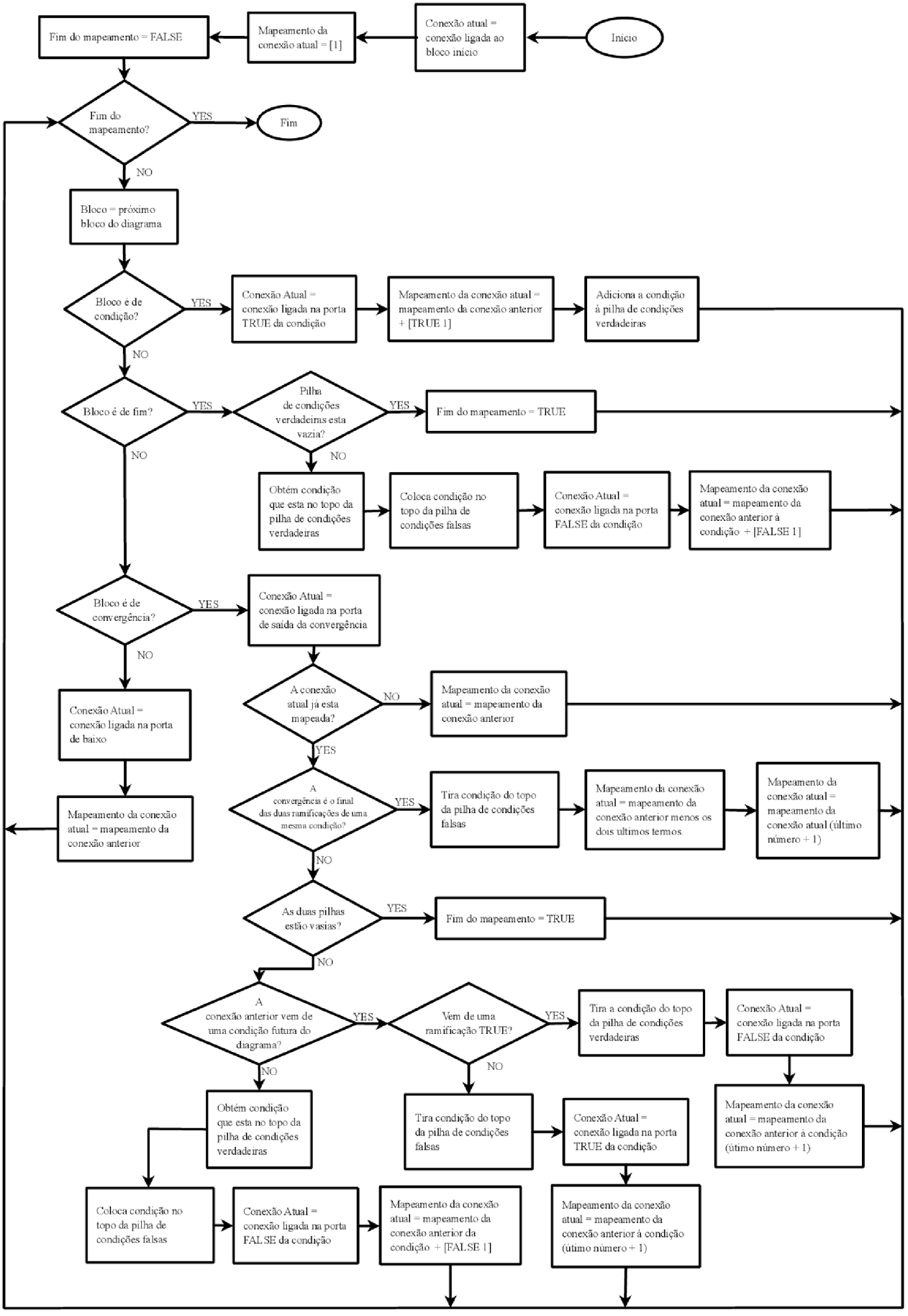



Figura 28 – Fluxograma do algoritmo de mapeamento.

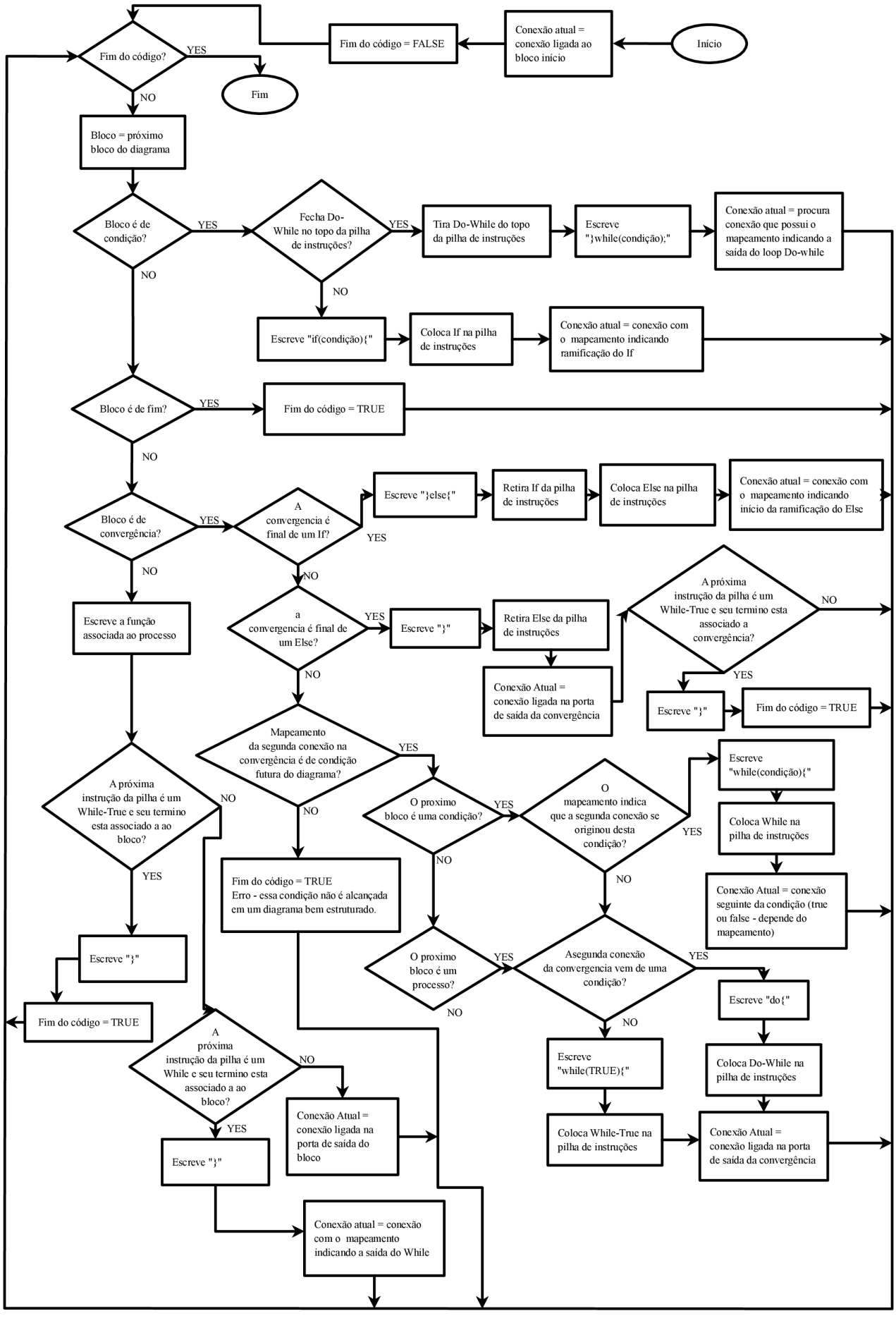



Figura 29 – Fluxograma do algoritmo de geração do código

# 6 Estudo de caso

Este capítulo tem por objetivo apresentar dois estudos de caso que foram realizados utilizando o *software* MBDAUTO. Cada um deles têm o intuito de demonstrar e validar a implementação da arquitetura aqui proposta. Nas seções seguintes serão apresentadas as ECUs e os dois estudos de caso realizados, descrevendo a metodologia utilizada e os resultados obtidos.

## 6.1 ECUs utilizadas

Para este estudo foram utilizadas duas placas diferentes para simular cada ECU, a AT89STK-06 *Demo Board* da Atmel e outra construída pelo autor. Para cada uma delas as funções presentes no *basic software* foram desenvolvidas separadamente, de acordo com suas configurações de hardware.

A placa AT89STK-06 (Fig. 30) possui o microcontrolador T89C51CC01 e dispõe de um controlador de comunicação serial, um controlador de comunicação CAN, um potenciômetro associado a um conversor A/D, sensor de temperatura e uma saída digital.

Figura 30 – Placa AT89STK-06 que simula uma das ECUs utilizadas.

Fonte: <http://www.atmel.com/tools/AT89STK-06CAN.aspx> disponível em 15 de setembro de 2014.

A segunda placa (Fig. 31) foi construída com o micro-controlador PIC-16F877A, nele são utilizados duas entradas analógicas e uma entrada de interrupção externa. Este micro-controlador não possui controlador CAN, e portanto precisou ser implementado em um circuito externo a partir do controlador MCP2515 que é gerenciado pelo PIC por meio da interface serial.

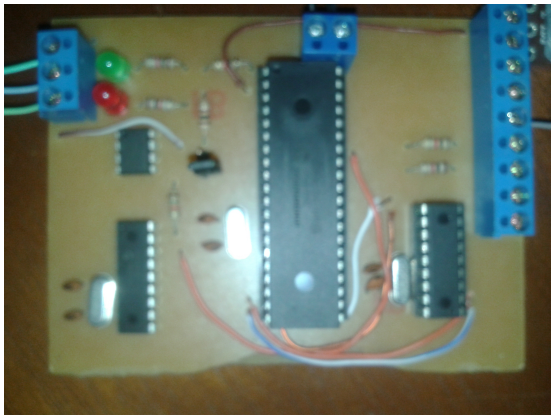


Figura 31 – Placa PIC-16F877A que simula uma das ECUs utilizadas.

## 6.2 Caso 01: Leitura da temperatura do óleo do motor

O primeiro estudo de caso consistiu da implementação de um sistemas composto por duas ECUs se comunicando através do barramento CAN. Onde a primeira ECU simula o funcionamento do ECM (*Engine Control Module*) que é responsável por monitorar e controlar informações presentes nos veículos.

Para o caso aqui implementado o ECM é responsável por fazer a leitura de um sensor de temperatura e enviar este dado para a rede de comunicação do veículo. Já a segunda ECU executa a função do IPC (*Instrument Panel Cluster*), que tem por finalidade controlar os dados mostrados no painel de instrumentos. O esquemático do sistema proposto está apresentado na Fig. 32.

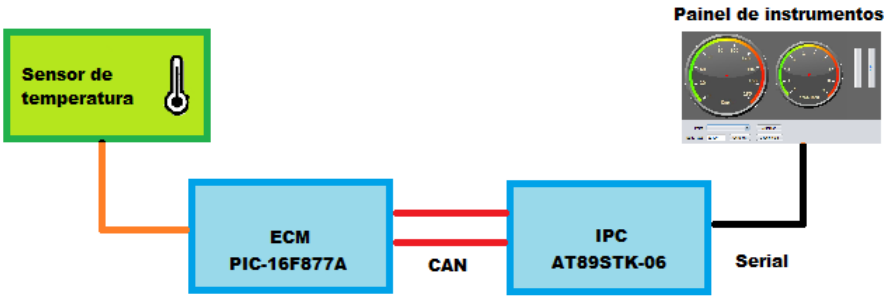



Figura 32 – Esquemático do caso 01.

No caso aqui apresentado o ECM foi simulado por meio da placa PIC-16F877A sendo que o potenciômetro presente na placa foi utilizado no lugar do sensor de temperatura. Já o IPC foi simulado através da placa AT89STK-06 da Atmel. Com o intuito simular o painel de instrumentos foi desenvolvido um programa em Java que apresenta os dados recebidos pela porta serial. Para isso o RTE fornece serviços de acesso a este painel (Fig. 33).

Para modelar este sistema no *software* MBDAUTO primeiramente foi construído o diagrama com as duas ECUs, conforme é apresentado na Fig. 34. A partir dele é possível


Figura 33 – Painel de instrumentos desenvolvido em Java.

construir o diagrama de fluxo de dados e o diagrama de atividades para cada uma das duas placas.

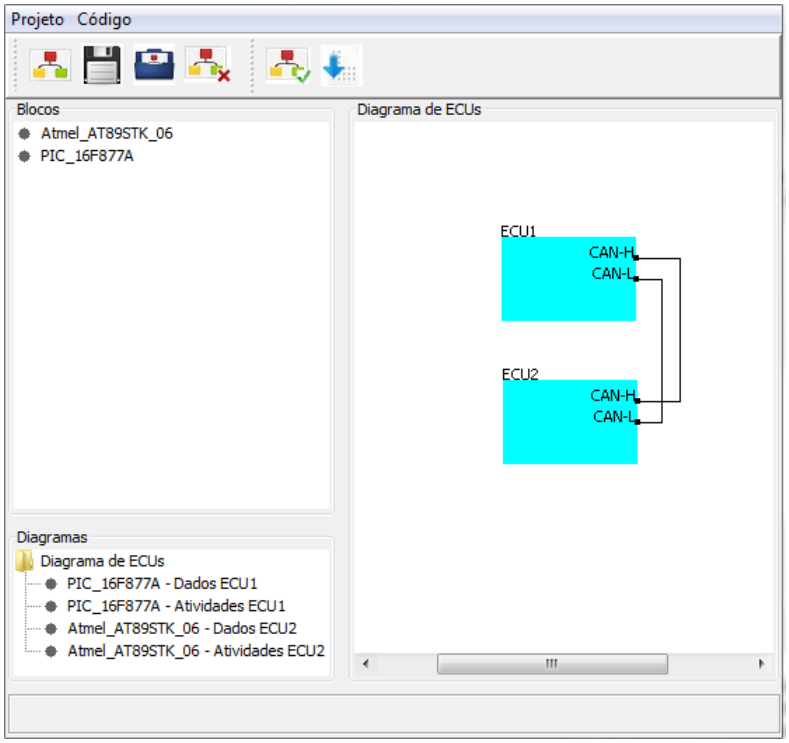



Figura 34 – Caso 01 - Diagrama de ECUs.

### 6.2.1 *Software* de leitura da temperatura

Para implementar o ECM utilizando o MBDAUTO é necessário construir os dois diagramas referentes à função que se deseja executar. O primeiro diagrama a ser construído é o de fluxo de dados (Fig. 35), nele o sistema é representado indicando o caminho que

os sinais devem percorrer. Em seu desenvolvimento foi utilizado um bloco de leitura do sensor, um de empacotamento J1939 e um de envio do *frame* CAN.

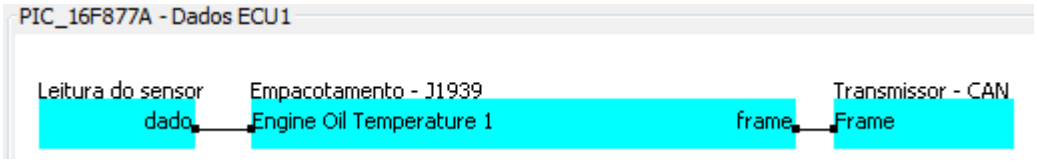



Figura 35 – Caso 01 - Diagrama de fluxo de dados do ECM.

Cada um destes blocos estão relacionados a serviços presentes na camada do RTE. A partir deles é gerado um arquivo de configuração que especifica quais serviços são utilizados. Como pode ser visto abaixo este arquivo consiste de um conjunto de *defines* indicando que esta aplicação usara as funções relacionadas ao controlador CAN, à norma J1939 e de leitura de sensores.

```
#define SENSORES
#define J1939
#define ET1
#define CAN
#define CAN_TX
```

O segundo diagrama a ser constuído é o de atividades (Fig. 36), nele deve estar descrita a lógica que será implementada na ECU. Em sua construção devem ser utilizadas as funções necessárias para configurar os recursos da placa e para descrever o comportamento esperado da ECU.

A primeira função posicionada no diagrama é responsável por configurar o microcontralador de acordo com suas ligações de hardware, em seguida o controlador CAN é configurado em seu estado inicial e é ligado. Após os ajustes iniciais, as funções que serão executadas permanentemente são posicionadas dentro de um laço infinito, sendo que elas realizam a leitura do sensor e empacotam o dado obtido para o *frame* CAN.

A função de leitura do sensor apresentada no diagrama fornece um serviço genérico, onde é necessário especificar em seu argumento de entrada o identificador do dado que precisa ser lido. Este identificador é enviado ao *software* básico, que verifica se o *hardware* possui o sensor especificado. Após obter o dado do sensor, ele é utilizado como argumento da função *update_data*, que empacota o dado de acordo com o seu identificador. Para todos os serviços implementados neste trabalho os identificadores são os mesmos apresentados para cada dado da norma J1939.

Neste diagrama não é necessário colocar uma função de envio do *frame* CAN, pois a biblioteca do RTE possui um serviço que controla a taxa de transmissão dos pacotes de acordo com o que é especificado pela norma J1939.

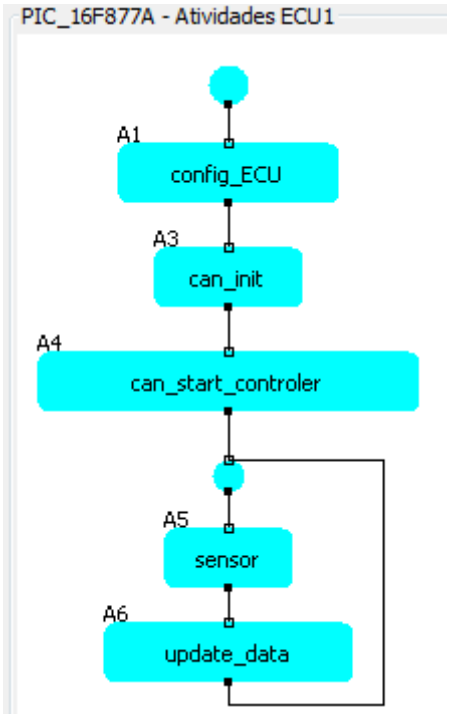



Figura 36 – Caso 01 - Diagrama de atividades do ECM.

Por meio deste diagrama o código destinado a executar esta aplicação é gerado automaticamente pelo MBDAUTO utilizando as funções presentes no RTE, o resultado é apresentado abaixo.

```
#include "config.h"

void main(){
  int16 temp;

  config_ECU();
  can_init();
  can_start_controler();
  while(1){
    temp = sensor(175);
    update_data(175,temp);
  }
}
```

### 6.2.2 *Software* do painel de instrumentos

Para implementar o IPC utilizando o MBDAUTO, deve-se construir os dois diagramas referentes a função que se espera executar. O primeiro diagrama a ser construído é o de fluxo de dados (Fig. 37). Nele foi utilizado um bloco receptor CAN, um para

desempacotar a mensagem recebida e o bloco do painel de instrumentos.

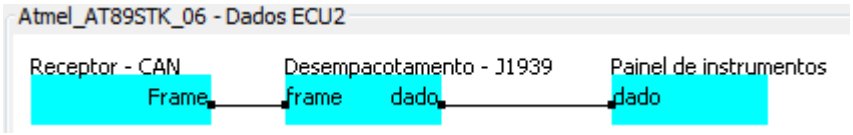



Figura 37 – Caso 01 - Diagrama de fluxo de dados do IPC.

Cada um destes blocos estão relacionados a serviços presentes na camada do RTE. A partir deles é gerado um arquivo de configuração que especifica quais serviços são utilizados. Como pode ser visto abaixo este arquivo consiste de um conjunto de *defines* indicando que esta aplicação usara as funções relacionadas ao controlador CAN, à norma J1939 e de acesso ao painel de instrumentos.

```
#define CAN
#define CAN_RX
#define J1939
#define PAINEL
```

O segundo diagrama a ser construído é o de atividades (Fig. 38), nele deve estar descrita a lógica que será implementada na ECU. Em sua construção devem ser utilizadas as funções necessárias para configurar os recursos da placa e para descrever o comportamento esperado da ECU.

As duas primeiras funções posicionadas no diagrama são responsáveis por configurar o controlador CAN e o de comunicação serial, que é necessário para os serviços de acesso ao painel. Para a inicialização da comunicação serial, deve-se configurar o valor da taxa de transmissão através do argumento da função *uart_init*.

Após os ajustes iniciais o controlador CAN é ligado e então a máscara e filtro que seleciona qual mensagem pode ser recebida é configurada. Para isso utiliza-se o bloco da função *can_set_filt_and_mask*, que recebe como argumentos o valor do identificador e uma informação a respeito do seu formato. Para encontrar o valor deste identificador foi utilizada a função *can_id*, que é um serviço da J1939 responsável por construir o identificador de acordo com os parâmetros apresentados na norma.

As funções que serão executadas permanentemente são posicionadas dentro de um laço infinito. A primeira função dentro dele é a *can_getd*, que recebe como argumentos um conjunto de endereços aonde devem ser armazenadas as informações caso haja recepção. Se alguma mensagem for recebida, esta função retorna verdadeiro.

Em seguida é utilizado um bloco de decisão que testa o retorno da função anterior, dessa maneira se houver recepção a função *get_data_J1939* é executada. Ela recebe como argumentos o identificador do frame, o identificador do dado a ser obtido, o endereço

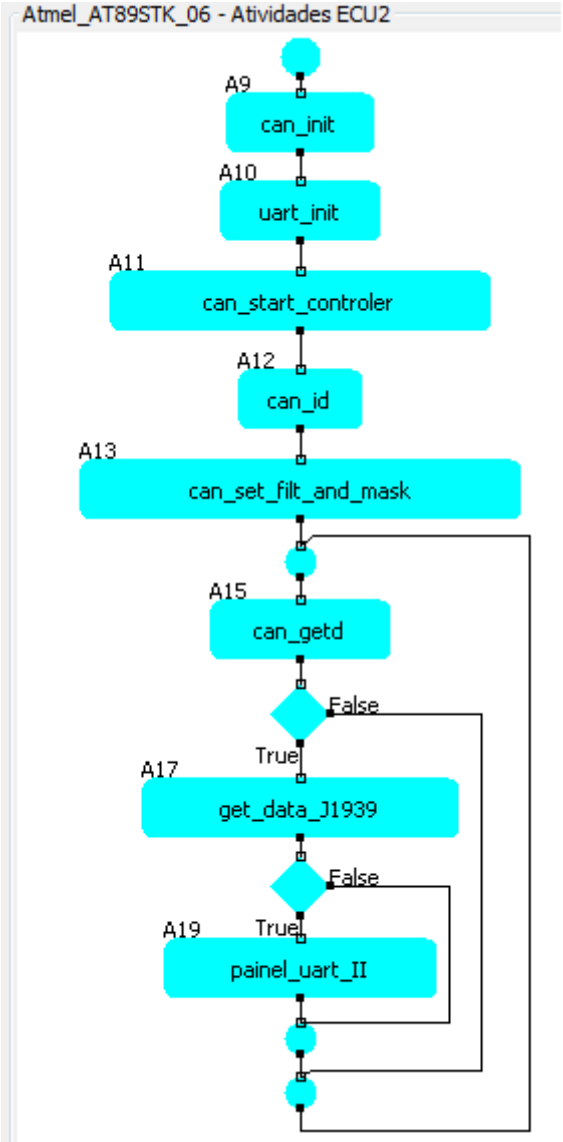



Figura 38 – Caso 01 - Diagrama de atividades do IPC.

aonde está armazenado o *frame* e o endereço no qual deve ser armazenado o dado pedido, ela retorna verdadeiro se o dado requerido estiver neste *frame*. Este valor retornado é testado por outro bloco de decisão, dessa maneira se o dado esperado tiver sido obtido corretamente a função de acesso ao painel é executada, sendo que ela tem como argumentos o dado e o identificador deste dado.

Por meio deste diagrama, o código destinado a executar a aplicação do IPC é gerado automaticamente pelo MBDAUTO utilizando as funções presentes no RTE. O resultado é apresentado abaixo.

```
#include "config.h"

void main(){
  int32 id;
  int1 status;
```

```
  int32 id_frame;
  int8 msg[8];
  int8 len;
  struct rx_stat status_frame;
  int16 dado;

  can_init();
  uart_init(38400);
  can_start_controler();
  id = can_id(3,65262,100);
  can_set_filt_and_mask(id,1);
  while(1){
    status = can_getd(&id_frame,&msg[0],&len,&status_frame);
    if(status){
      status = get_data_J1939(id_frame,175,&msg[0],&dado);
      if(status){
        painel_uart_II(dado,175);
      }else{
      }
    }else{
    }
  }
}
```

### 6.2.3 Análise dos resultados

A partir dos dois códigos aqui apresentados o sistema com as duas ECUs foi implementado (Fig. 39). Dessa maneira pôde-se visualizar seu funcionamento, onde a variação do potenciômetro no ECM alterava o indicador de temperatura no painel de instrumentos.

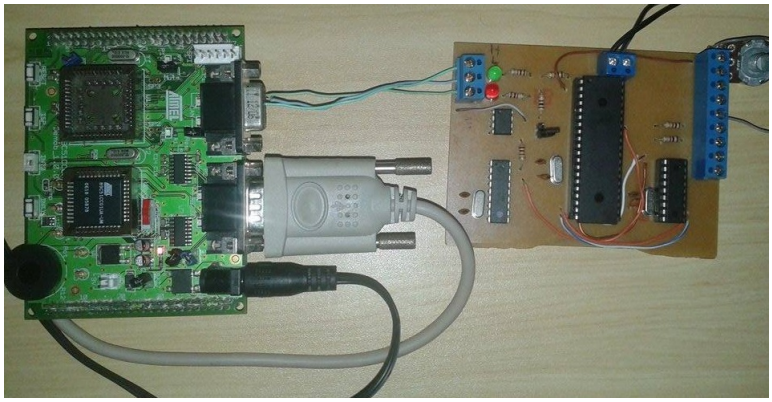


Figura 39 – Circuito implementado para o caso 01.

Na Figura 40 é possível visualizar a indicação da temperatura e também todos os dados transmitidos por meio da comunicação serial. Nota-se que além dos *frames* recebidos pela ECU, também é impressa a informação que é passada ao painel de instrumentos, essa informação é composta do valor da temperatura é do identificador do dado da temperatura, dessa maneira o painel consegue identificar qual dos seus mostradores precisa ser atualizado.

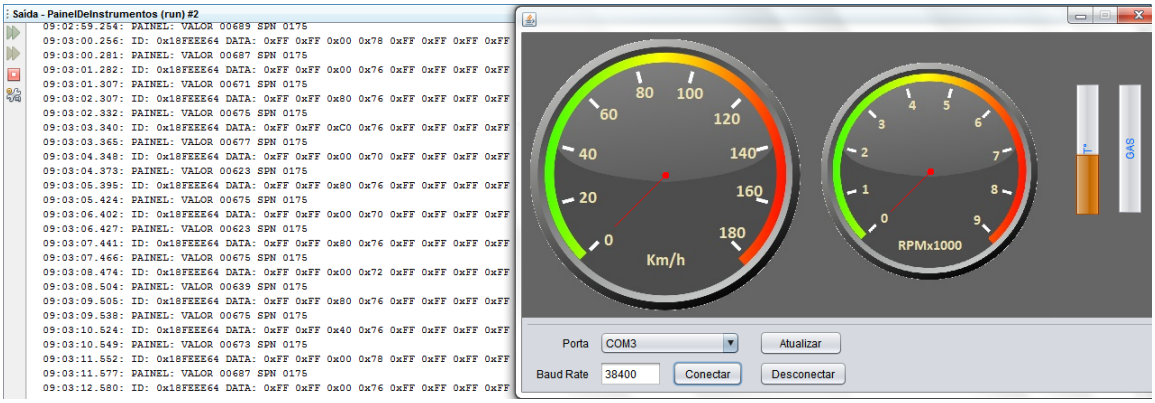



Figura 40 – Painel de instrumentos indicando variação da temperatura.

Posteriormente foi coletado somente os *frames* recebidos pela ECU conforme é apresentado na Fig. 41. Por meio desta informação é possível verificar que a mensagem da temperatura, posicionada na terceira e quarta posição da mensagem, é o único dado sendo transmitido no *frame* e que ela é enviada a cada um segundo conforme o que é especificado na norma J1939.

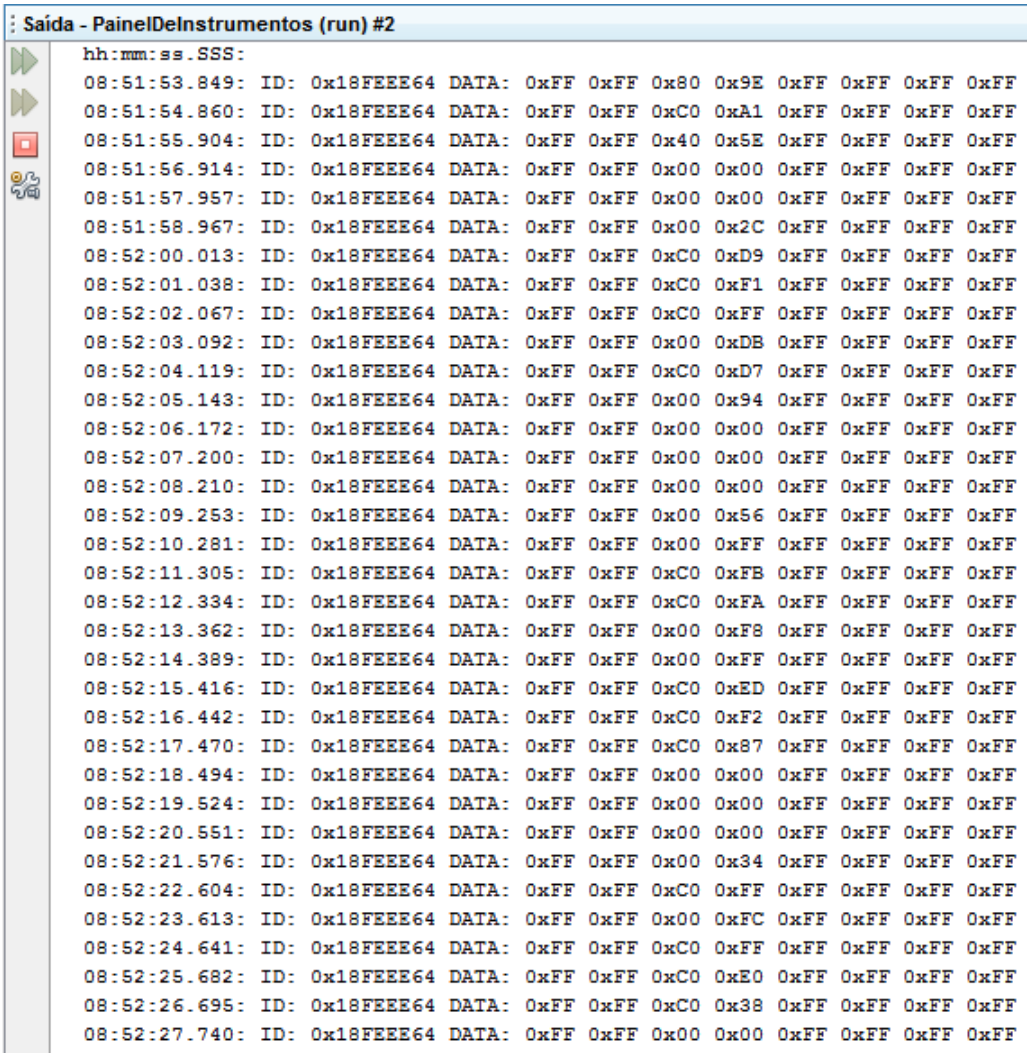



Figura 41 – *Frames* recebidos pelo barramento CAN.

## 6.3 Caso 02: Leitura da velocidade e rotação do motor

O segundo estudo de caso consistiu da implementação de um sistemas composto por duas ECUs se comunicando através do barramento CAN. Onde a primeira ECU simula o funcionamento do ECM (*Engine Control Module*) que é responsável por monitorar e controlar diferentes informações presentes nos veículos.

A diferença básica é que, para o caso aqui implementado, o ECM é responsável por fazer a leitura de dois sensores, um de rotação e um de velocidade, e enviar este dados para a rede de comunicação do veículo. Já a segunda ECU executa a função do IPC (*Instrument Panel Cluster*), que tem por finalidade controlar os dados mostrados no painel de instrumentos. O esquemático do sistema proposto esta na Fig. 42.

No caso aqui apresentado as duas ECUs foram simuladas utilizando a placa AT89STK-06 da Atmel. Como esta placa dispõe de somente um conversor ADC, o serviço de leitura

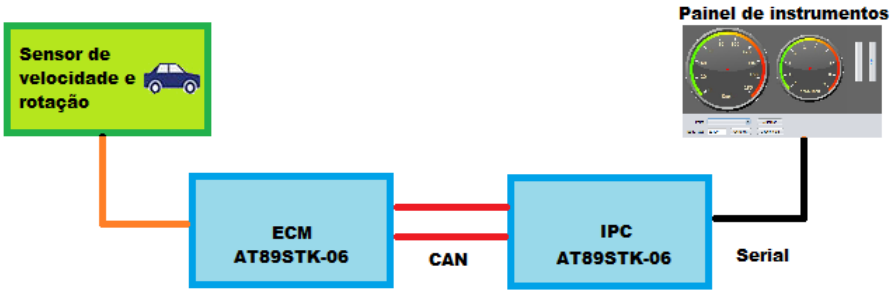



Figura 42 – Esquemático do caso 02.

dos sensores foi realizado pelo *software* básico por meio da comunicação serial, sendo que para isso foi desenvolvida uma aplicação em Java que permite controlar os valores que são enviados em resposta a cada sensor (Fig. 43). Para simular o painel de instrumentos foi utilizado o programa em Java que foi apresentado no primeiro estudo de caso (Fig. 33) para o qual o RTE fornece serviços de controle.

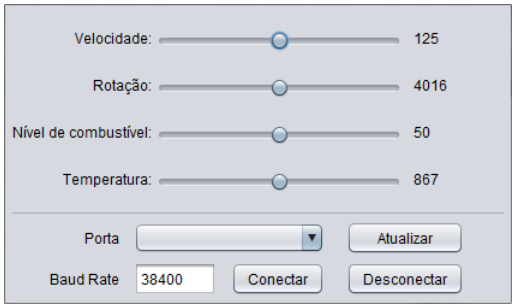



Figura 43 – Programa para simular a leitura dos sensores.

Para modelar este sistema no *software* MBDAUTO primeiramente foi construído o diagrama com as duas ECUs, conforme é apresentado na Fig. 44. A partir dele é possível construir os diagramas referentes ao funcionamento de cada uma das duas placas.

### 6.3.1 *Software* de Leitura de velocidade e de rotação

Para implementar o ECM utilizando o MBDAUTO é necessário construir os dois diagramas referentes à função que se espera executar. O primeiro diagrama a ser construído é o de fluxo de dados (Fig. 45), nele o sistema é representado indicando o caminho que os sinais devem percorrer. Em seu desenvolvimento foram utilizados dois blocos de leitura do sensor, um de empacotamento J1939 e um de envio do *frame* CAN.

Cada um destes blocos estão relacionados a serviços presentes na camada do RTE. A partir deles é gerado um arquivo de configuração que especifica quais serviços são utilizados. Como pode ser visto abaixo este arquivo consiste de um conjunto de *defines* indicando que esta aplicação usara as funções relacionadas ao controlador CAN, à norma J1939 e à leitura dos sensores.

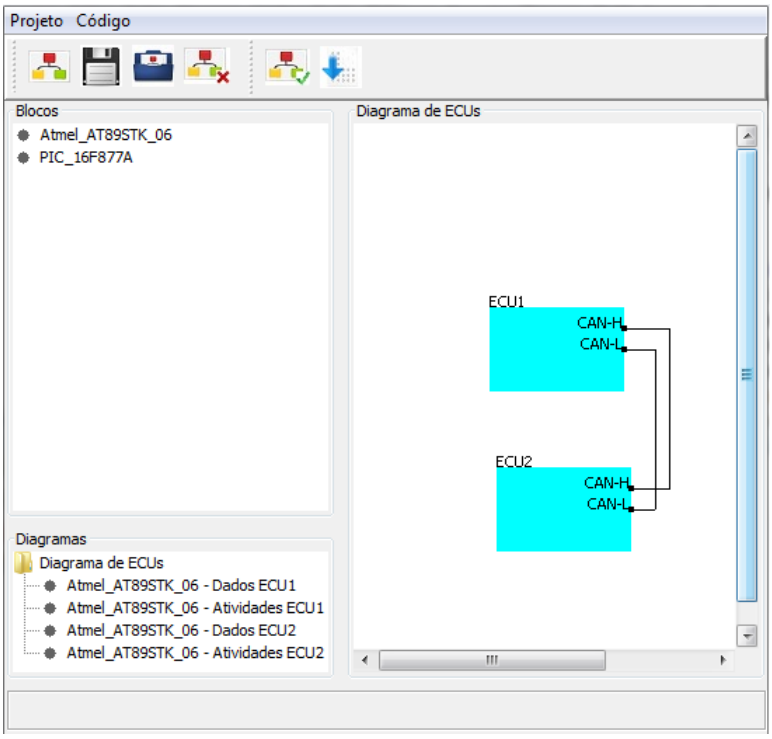



Figura 44 – Caso 02 - Diagrama de ECUs.

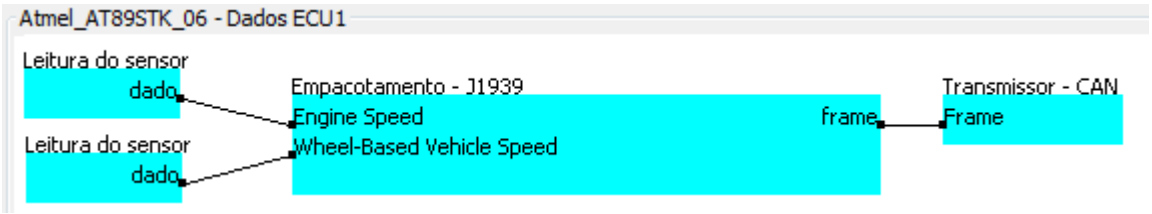



Figura 45 – Caso 02 - Diagrama de fluxo de dados do ECM.

```
#define SENSORES
#define J1939
#define EEC1
#define CCVS
#define CAN
#define CAN_TX
```

Quanto ao diagrama de atividades (Fig. 46), nele deve estar descrita a lógica que será implementada na ECU. Em sua construção devem ser utilizadas as funções necessárias para configurar os recursos da placa e para descrever o comportamento esperado da ECU.

A primeira função presente no diagrama é responsável por configurar o controlador de acordo com suas ligações de hardware, em seguida os controladores do CAN e da comunicação serial são configurados em seu estado inicial e inicializados. Após estes ajustes iniciais, as funções que serão executadas permanentemente são posicionadas dentro de um

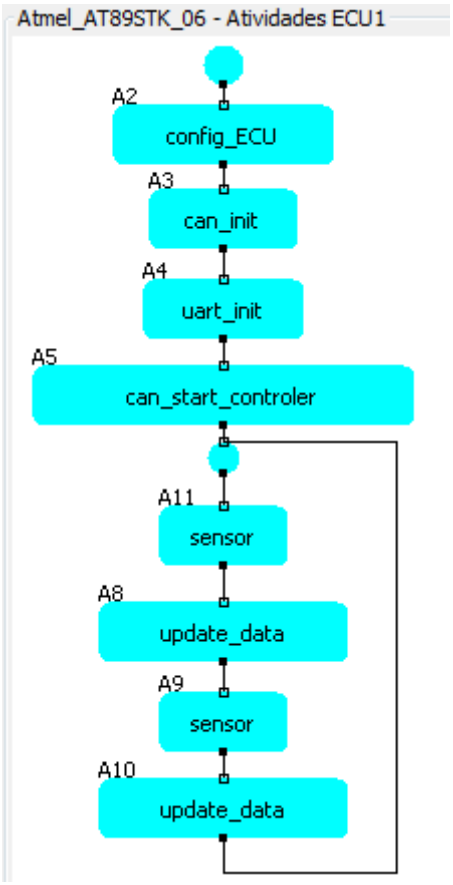



Figura 46 – Caso 02 - Diagrama de atividades do ECM.

laço infinito, sendo que elas realizam a leitura do sensor e empacotam o dado obtido para o *frame* CAN. Este procedimento de acesso ao sensor e empacotamento é feito duas vezes, uma para ler a informação com o identificador da rotação e a outra para ler a informação da velocidade.

A função de leitura do sensor apresentada no diagrama fornece um serviço genérico, onde é necessário especificar em seu argumento de entrada o identificador do dado que precisa ser lido. Este identificador é enviado ao *software* básico, que verifica se o *hardware* possui o sensor especificado. Após obter o dado do sensor, ele é utilizado como argumento da função *update_data*, que empacota o dado de acordo com o seu identificador.

Neste diagrama não é necessário colocar uma função de envio, pois a biblioteca do RTE possui um serviço específico da norma J1939 que controla a taxa de transmissão dos pacotes de acordo com o que é descrito pela norma.

Após a construção dos diagramas o código para executar esta aplicação foi gerado automaticamente pelo MBDAUTO utilizando as funções presentes no RTE, conforme é apresentado abaixo.

```
#include "config.h"
```

```
void main(){
  int16 rotacao;
  int16 velocidade;

  config_ECU();
  can_init();
  uart_init(38400);
  can_start_controler();
  while(1){
    rotacao = sensor(190);
    update_data(190,rotacao);
    velocidade = sensor(84);
    update_data(84,velocidade);
  }
}
```

### 6.3.2 *Software* do painel de instrumentos

Para implementar o IPC utilizando o MBDAUTO deve-se construir os dois diagramas referentes a função que se espera executar. O primeiro diagrama a ser construído é o de fluxo de dados (Fig. 47). Nele foi utilizado um bloco do receptor CAN, um para desempacotar a mensagem recebida e o bloco do painel de instrumentos.

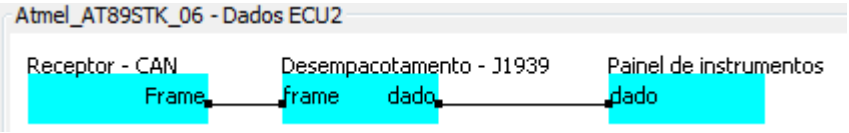



Figura 47 – Caso 02 - Diagrama de fluxo de dados do IPC.

Cada um destes blocos estão relacionados a serviços presentes na camada do RTE. A partir deles é gerado um arquivo de configuração que especifica quais serviços são utilizados. Como pode ser visto abaixo este arquivo consiste de um conjunto de *defines* indicando que esta aplicação usara as funções relacionadas ao controlador CAN, à norma J1939 e de acesso ao painel de instrumentos.

```
#define CAN
#define CAN_RX
#define J1939
#define PAINEL
```

O segundo diagrama a ser constuído é o de atividades (Fig. 48), nele deve estar descrita a lógica que será implementada na ECU. Em sua construção devem ser utilizadas

as funções necessárias para configurar os recursos da placa e para descrever o comportamento esperado da ECU.

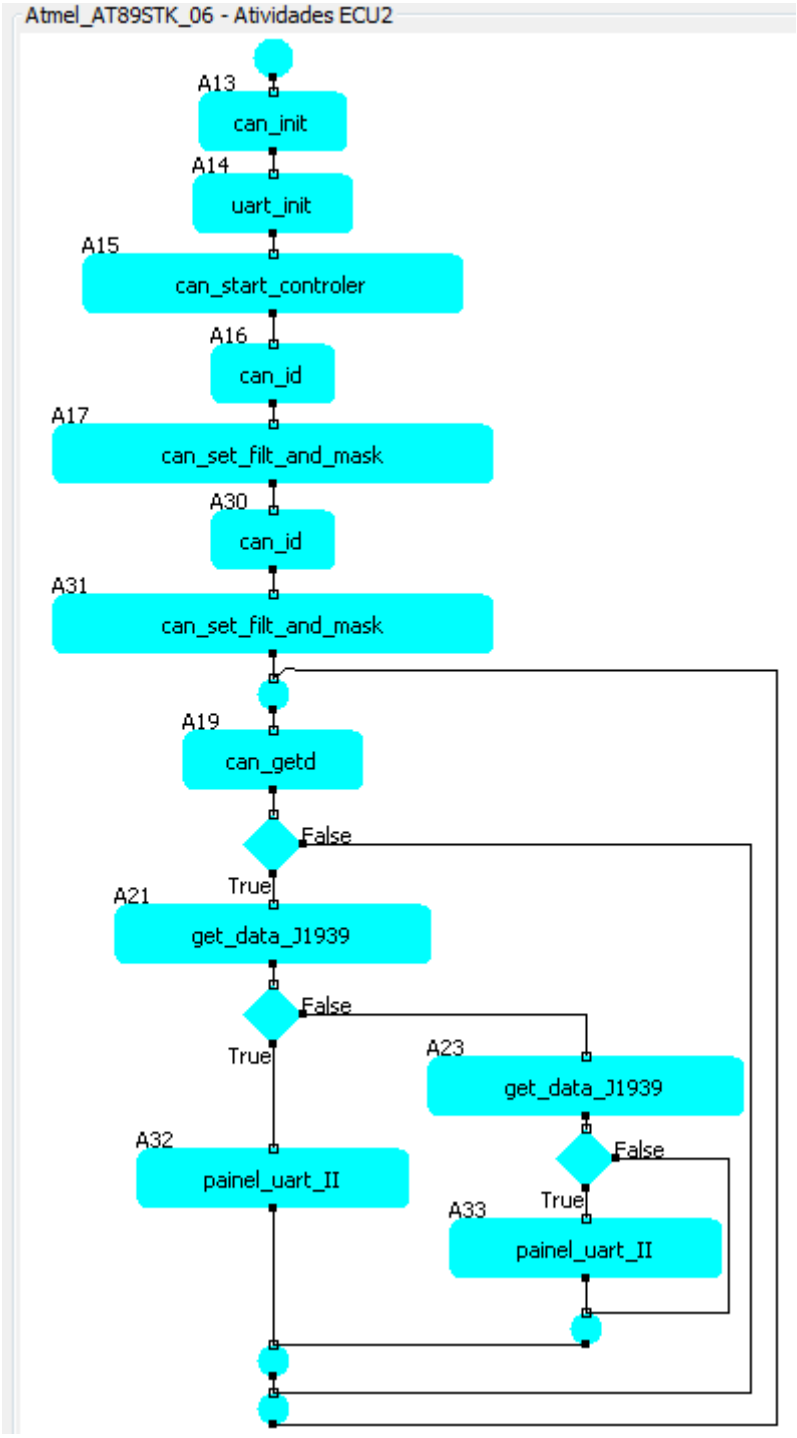



Figura 48 – Caso 02 - Diagrama de atividades do IPC.

As duas primeiras funções posicionadas no diagrama são responsáveis por configurar o controlador CAN e o de comunicação serial, que é necessário para os serviços de acesso ao painel. Para a inicialização da comunicação serial, deve-se configurar o valor da taxa de transmissão através do argumento da função *uart_init*.

Após os ajustes iniciais o controlador CAN é ligado e então as máscaras e filtros que selecionam qual mensagem pode ser recebida é configurada. Para isso utiliza-se o bloco da função *can_set_filt_and_mask*, que recebe como argumentos o valor do identificador do *frame* e uma informação a respeito do seu formato. Para encontrar o valor deste identificador foi utilizada a função *can_id*, que é um serviço da J1939 responsável por construir o identificador de acordo com os parâmetros apresentados na norma. Para configurar as máscaras e filtros para receber mais de um identificador, basta chamar esta função sequencialmente, especificando um identificador por vez.

As funções que serão executadas permanentemente são posicionadas dentro de um laço infinito. A primeira função dentro dele é a *can_getd*, que recebe como argumentos o conjunto de endereços aonde devem ser armazenadas as informações caso haja recepção. Se alguma mensagem for recebida, esta função retorna verdadeiro.

Em seguida é utilizado um bloco de decisão que testa o retorno da função anterior, dessa maneira se houver recepção a função *get_data_J1939* é executada. Ela recebe como argumentos o identificador do frame, o identificador do dado a ser obtido, o endereço aonde está armazenado o *frame* e o endereço no qual deve ser armazenado o dado pedido, ela retorna verdadeiro se o dado requerido estiver neste *frame*.

Este valor retornado é testado por outro bloco de decisão, dessa maneira se o dado esperado tiver sido obtido corretamente a função de acesso ao painel é executada, sendo que ela tem como argumentos o dado e o identificador deste dado. Este procedimento é realizado duas vezes, primeiro o programa tenta extrair a informação de rotação, caso este não seja o *frame* correto, ele tenta extrair a informação de velocidade. Quando o programa encontra o dado correto ele é enviado ao painel.

Após a construção dos diagramas o código destinado a executar esta aplicação foi gerado automaticamente pelo MBDAUTO utilizando as funções presentes no RTE, conforme é apresentado abaixo.

```
#include "config.h"

void main(){
  int32 id;
  int1 status;
  int32 id_frame;
  int8 msg[8];
  int8 len;
  struct rx_stat status_frame;
  int1 statusRotacao;
  int16 dado;
  int1 statusVelocidade;

  can_init();
```

```
  uart_init(38400);
  can_start_controler();
  id = can_id(3,61444,100);
  can_set_filt_and_mask(id,1);
  id = can_id(6,65265,100);
  can_set_filt_and_mask(id,1);
  while(1){
    status = can_getd(&id_frame,&msg[0],&len,&status_frame);
    if(status){
      statusRotacao = get_data_J1939(id_frame,190,&msg[0],&dado);
      if(statusRotacao){
        painel_uart_II(dado,190);
      }else{
        statusVelocidade = get_data_J1939(id_frame,84,&msg[0],&dado);
        if(statusVelocidade){
          painel_uart_II(dado,84);
        }else{
        }
      }
    }else{
    }
  }
}
```

### 6.3.3 Análise dos resultados

A partir dos dois códigos aqui apresentados o sistema com o ECM e o IPC foi implementado (Fig. 49). Dessa maneira pôde-se visualizar seu funcionamento, de modo que o painel de instrumentos mostrou adequadamente a variação dos parâmetros a medida que o ajuste era efetuado.

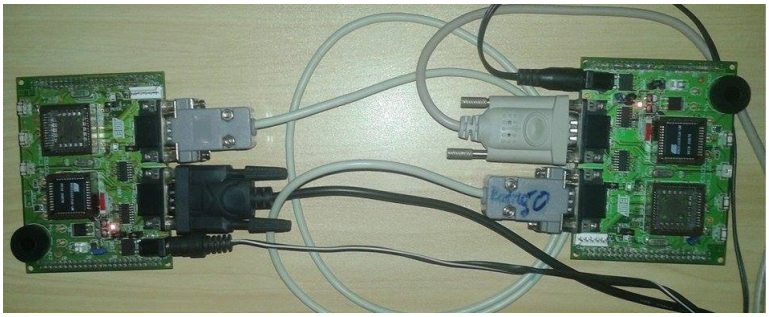


Figura 49 – Circuito implementado para o caso 02.

Na Figura 50 é possível visualizar a indicação da velocidade e da rotação além de todos os dados transmitidos por meio da comunicação serial. Nota-se que além dos *frames*

recebidos pela ECU, também é apresentada a informação que é enviada ao painel de instrumentos, essa informação é composta do valor e do identificador da mensagem. Como pode ser visto na figura o valor da rotação e enviado juntamente com o identificador 190 e o valor da velocidade com o identificador 84. Dessa maneira o painel consegue identificar qual dos seus mostradores precisa ser atualizado.

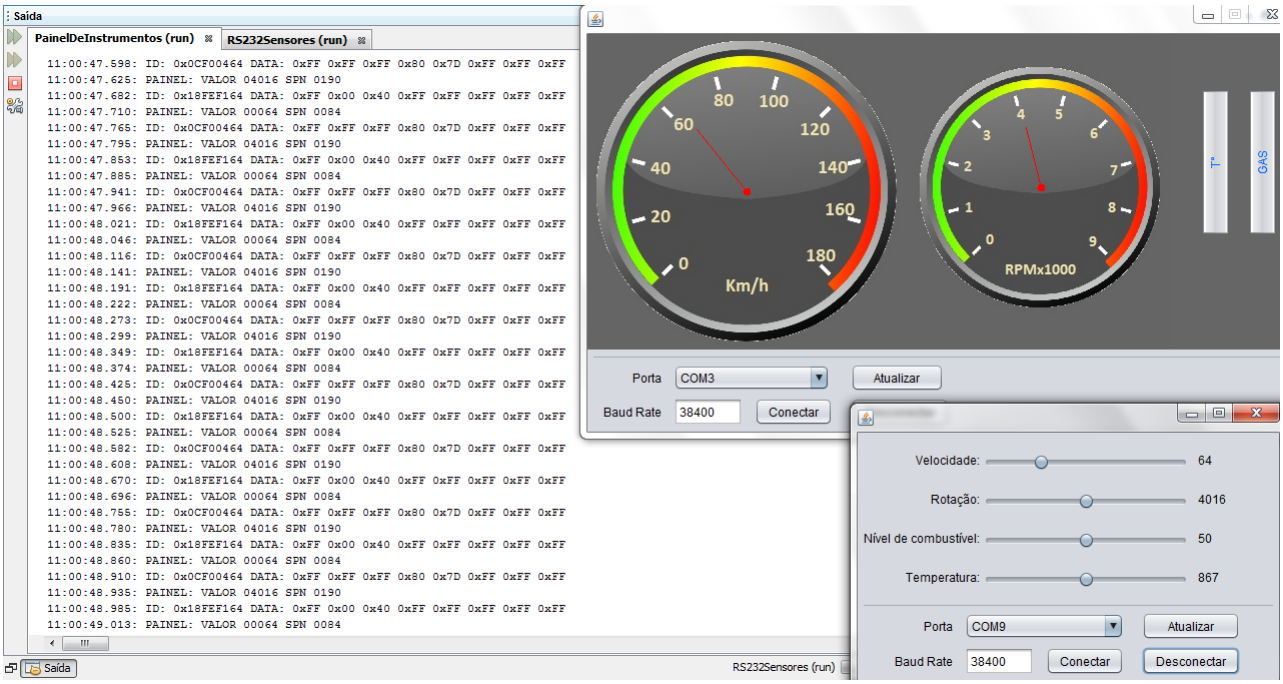



Figura 50 – Painel de instrumentos indicando variação de rotação e de velocidade.

Em um segundo momento foi coletado somente os *frames* recebidos pela ECU, conforme é apresentado na Fig. 51. Nela é possível identificar dois *frames* com identificadores diferentes, um que armazena a velocidade nos *bytes* dois e três e possui o identificador 0x18FEF164 e o outro que armazena a rotação nos *bytes* quatro e cinco e apresenta o identificador 0x0CF00464.

Nesta figura também é possível perceber que os dois *frames* são recebidos intercalados, o que ocorre devido aos dois estarem configurados com a mesma taxa de transmissão que é de 100ms. Por meio desta figura pode-se verificar que os dados apresentados estão com uma taxa de recepção bem próxima ao que era esperado.

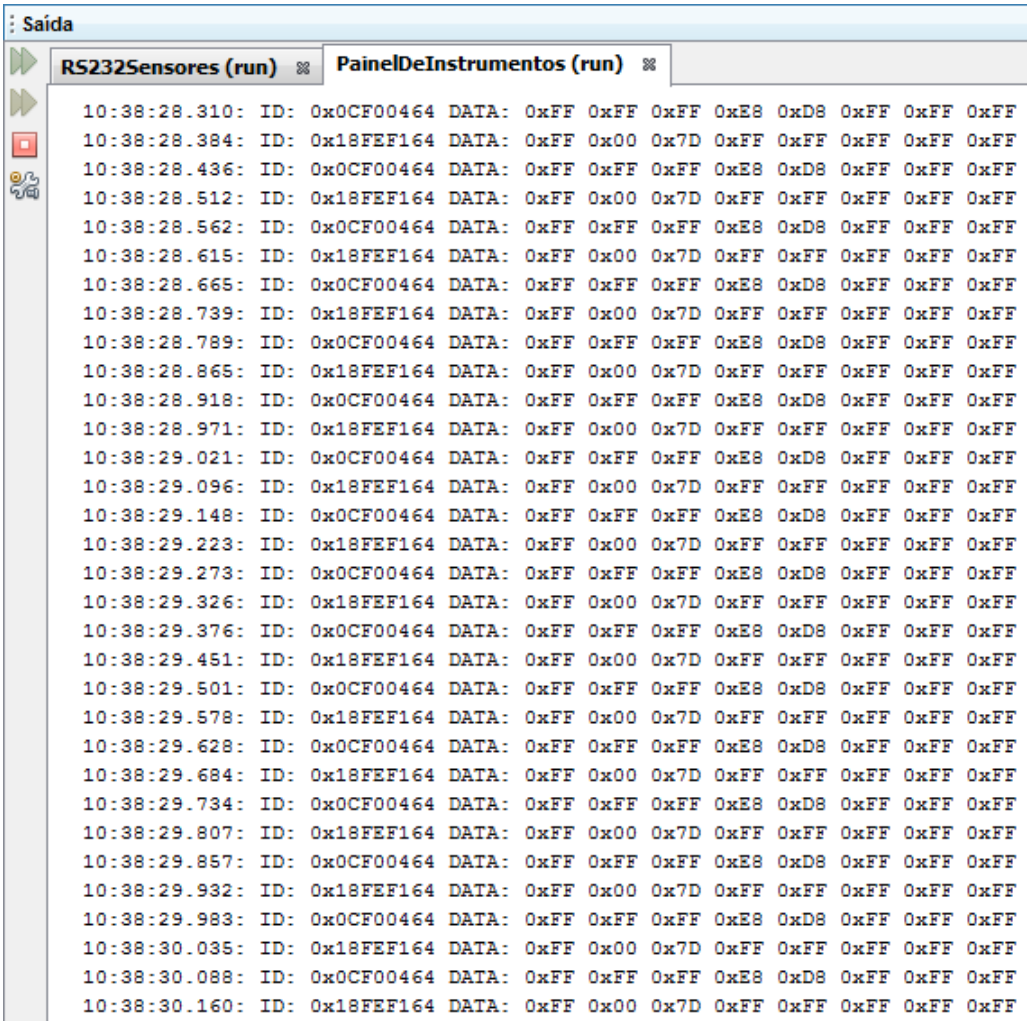



Figura 51 – *Frames* recebidos pelo barramento CAN.

Desta maneira percebe-se que o sistema operou em concordância com o que foi projetado e que ele atendeu as especificações de taxa de transmissão e de empacotamento da mensagem. Além disso o resultado obtido demonstrou que a metodologia de desenvolvimento baseada na modelagem das funcionalidades do sistema é capaz de reduzir de forma significativa o esforço envolvido na implementação de novos casos, uma vez que o desenvolvimento foi concentrado na descrição do comportamento do sistema através dos diagramas. Além disso, o *software* básico de cada ECU precisa ser desenvolvido apenas uma vez e os serviços presentes no RTE podem ser reutilizados para qualquer *hardware* escolhido.

# 7 Conclusão

Devido aos problemas relacionados à crescente complexidade e a falta de padronização dos *softwares* automotivos, surgiu a necessidade de desenvolver especificações e metodologias que acelerem o seu desenvolvimento. Diante disso os principais envolvidos na indústria automotiva propuseram o padrão AUTOSAR, que procura especificar uma arquitetura de *software* aberta e padronizada para proporcionar uma infraestrutura básica que auxilie no desenvolvimento de *software* embarcado automotivo.

Porém devido a falta de estudos que mostrem a eficiência em adotar a metodologia proposta pelo padrão, este esforço ainda não atingiu os benefícios que eram esperados. Uma vez que, mesmo o padrão estando definido e especificado, os fabricantes ainda estão receosos em investir nele.

Diante disso, esse trabalho apresentou uma arquitetura de desenvolvimento de *software* embarcado automotivo aderente ao padrão AUTOSAR. Abordando desde a estrutura das camadas da arquitetura até uma metodologia de desenvolvimento baseada na modelagem funcional do sistema. Para avaliar o funcionamento e a eficiência da arquitetura proposta foram realizados estudos de caso, sendo que eles demonstraram o funcionamento que era esperado do modelo apresentado.

O resultado obtido mostrou que a metodologia de desenvolvimento baseada na modelagem das funcionalidades do sistema é capaz de reduzir de forma significativa o esforço envolvido na implementação de novos casos, uma vez que o desenvolvimento é concentrado na descrição do comportamento do sistema através dos diagramas, e ocorre em um nível de abstração muito maior do que em metodologias tradicionais. Neste sentido, o esforço mais significativo da implementação desta arquitetura está na necessidade de se construir o *software* básico para cada nova ECU a ser utilizada.

Além disso, o uso desta padronização permite que os serviços implementados anteriormente possam ser reutilizados. Essa possibilidade reduz consideravelmente o tempo de desenvolvimento, e permite que os engenheiros se concentrem mais na tarefa de aprimorar e atualizar as funcionalidades ao invés de desenvolver todo o sistema novamente.

Ao longo do desenvolvimento do trabalho foi diagnosticado que o desafio mais significativo da implementação de uma padronização de *software* é a especificação de uma arquitetura que atenda a todos os recursos possíveis dos diferentes *hardwares*. Outro desafio consiste em desenvolver uma ferramenta integrada com a arquitetura e que seja eficiente para implementá-la.

Diante do que foi exposto, a principal contribuição deste trabalho se reflete em

apresentar o uso de padronizações como o AUTOSAR e também o desenvolvimento baseado em modelos, além de discutir quais as vantagens e desafios que esta metodologia pode trazer se utilizado pela indústria.

Como sugestão de trabalhos futuros, considera-se aprimorar o *software* MBDAUTO para operar utilizando o padrão AUTOSAR e melhorá-lo acrescentando novas funcionalidades como ferramentas de *debug* e de simulação. No que se refere ao algoritmo responsável por traduzir o diagrama de atividades considera-se desenvolver um algoritmo capaz de tratar qualquer fluxograma estando ele construído a partir das estruturas básicas ou não.

# Referências


ABOWD, P.; RUSHTON, G. *Extensible and Upgradeable Vehicle Electrical, Electronic, and Software Architectures*. [S.l.], 2002. Citado 2 vezes nas páginas 31 e 32.

BRITTO, R. d. S. Uma arquitetura distribuída de hardware e software para controle de um robô móvel autônomo. BR, 2008. Citado na página 23.

CARLISLE, M. C. et al. Raptor: introducing programming to non-majors with flowcharts. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 19, n. 4, p. 52–60, 2004. Citado na página 54.

COOK, J. A. et al. Control, computing and communications: technologies for the twenty-first century model t. *Proceedings of the IEEE*, IEEE, v. 95, n. 2, p. 334–355, 2007. Citado na página 21.

DAKHORE, H.; MAHAJAN, A. Generation of c-code using xml parser. *Proceedings of ISCET*, v. 2010, p. 19–20, 2010. Citado na página 54.

FENNEL, H. et al. Achievements and exploitation of the autosar development partnership. *Convergence*, v. 2006, p. 10, 2006. Citado 4 vezes nas páginas 35, 37, 38 e 39.

FONSECA, V. V. *Implementação de padronização AUTOSAR para arquitetura elétricas utilizadas em países emergentes*. 2013. Citado 2 vezes nas páginas 27 e 37.

GIMPEL, J. F. Contour: a method of preparing structured flowcharts. *ACM SIGPLAN Notices*, ACM, v. 15, n. 10, p. 35–41, 1980. Citado na página 53.

GUIMARÃES, A. A. *Eletrônica embarcada automotiva*. 1. ed. [S.l.]: Érica, 2007. Citado 6 vezes nas páginas 21, 22, 23, 25, 26 e 30.

GUIMARÃES, A. de A.; SARAIVA, A. M. O protocolo can: Entendendo e implementando uma rede de comunicação serial de dados baseada no barramento "controller area network". 2002. Citado na página 28.

JACKMAN, B.; SANYANGA, S. *A Software Component Architecture for Improving Vehicle Software Quality and Integration*. [S.l.], 2005. Citado 3 vezes nas páginas 19, 32 e 35.

JOHN, D. Osek/vdx history and structure. IET, 1998. Citado na página 33.

NAUMANN, N. Autosar runtime environment and virtual function bus. *Hasso-Plattner-Institut, Tech. Rep*, 2009. Citado na página 38.

OSEK-GROUP. *OSEK/VDX Binding Specification*. 2004. Citado 2 vezes nas páginas 33 e 34.

PARET, D. *Multiplexed networks for embedded systems: CAN, LIN, Flexray, Safe-by-Wire...* [S.l.]: John Wiley & Sons, 2007. Citado 3 vezes nas páginas 26, 27 e 28.

POLBERGER, D. *Component technology in an embedded system*. [S.l.]: Master's thesis in computer science, available online (http://www. polberger. se/components/). ISSN, 2009. Citado na página 36.

SANTOS, M. *REDES DE COMUNICAÇAO AUTOMOTIVA: CARACTERISTICAS, TECNOLOGIAS E APLICAÇOES*. [S.l.]: ERICA, 2010. ISBN 9788536502755. Citado na página 25.

SILVA, D.; BATISTA, I. J.; VARELA, T. Modelo de uma arquitetura de computação distribuída aplicada a robôs móveis. *IV Congresso de Pesquisa e Inovação da Rede Norte e Nordeste de Educação Tecnológica*, 2009. Citado na página 23.

SILVA, W. C. d. Gerência de interfaces para sistemas de informação: Uma abordagem baseada em modelos. Universidade Federal de Goiás, 2010. Citado na página 47.

SOUSA, F. M. de; ALENCAR, F. M.; CASTRO, J. O impacto dos cots no processo de engenharia de requisitos. In: *WER*. [S.l.: s.n.], 1999. p. 175–186. Citado na página 35.

VINCENTELLI, A. S. Automotive electronics: Trends and challenges. In: CITESEER. *SAE CONFERENCE PROCEEDINGS P*. [S.l.], 2000. p. 295–308. Citado na página 31.

VINCENTELLI, A. S.; NATALE, M. D. Embedded system design for automotive applications. *IEEE Computer*, v. 40, n. 10, p. 42–51, 2007. Citado 2 vezes nas páginas 19 e 31.

WU, X.-H. et al. Research and application of code automatic generation algorithm based on structured flowchart. *Journal of Software Engineering and Applications*, Scientific Research Publishing, v. 4, n. 09, p. 534, 2011. Citado na página 54.

YOO, S.; JERRAYA, A. A. Introduction to hardware abstraction layers for soc. In: *Embedded Software for SoC*. [S.l.]: Springer, 2003. p. 179–186. Citado na página 39.